Acoplamientos óptimos de caminos de longitud dos

Tesis presentada a la
Universidad Autónoma Metropolitana, Azcapotzalco
como requerimiento parcial para obtener el grado de
Maestro en Optimización
por

Ing. Gualberto Vazquez Casas

Asesores:

Dr. Marco Antonio Heredia Velasco
Departamento de Sistemas, UAM Azcapotzalco

Dr. Francisco Javier Zaragoza Martínez
Departamento de Sistemas, UAM Azcapotzalco

Ciudad de México, México
29 de agosto de 2017

Optimal Euclidean Non-Crossing 3-Matchings

A thesis presented to the
Universidad Autónoma Metropolitana, Azcapotzalco
in partial fulfillment of the requirements for the degree of
Master in Optimization
by

B.Eng. Gualberto Vazquez Casas

Thesis directors:
Ph.D. Marco Antonio Heredia Velasco
Departamento de Sistemas, UAM Azcapotzalco

Ph.D. Francisco Javier Zaragoza Martínez
Departamento de Sistemas, UAM Azcapotzalco

Mexico City, Mexico
August 29, 2017

Abstract

Let P be a set of $3k$ points in the Euclidean plane. A 3-matching is a partition of P into k subsets of 3 points each, called triplets. The cost of each triplet $\{a, b, c\}$ is given by $\min\{|ab| + |bc|, |bc| + |ca|, |ca| + |ab|\}$, and the cost of the 3-matching is the sum of the costs of its triplets. The Euclidean 3-matching problem consists on finding a minimum cost 3-matching of P under the Euclidean metric. In the usual formulation of the Euclidean 3-matching problem we need to find a minimum cost 3-matching of P. This problem has several applications, especially in the insertion of components on a printed circuit board. Johnsson, Magyar, and Nevalainen introduced two integer programming formulations for this problem, and proved that its decision version is NP-complete if each triplet has an arbitrary positive cost (i.e., not necessarily Euclidean). The problem remains NP-complete even if the points of P correspond to vertices of a unit distance graph (a metric cost function).

In this work, we prove that the linear programming relaxations of these two models are equivalent. Then we introduce three new integer programming models that use fewer variables than those from Johnsson, Magyar, and Nevalainen. We also compare the linear programming relaxations of the models. Besides the minimization problem, we are also interested in a similar maximization problem: finding a maximum cost non-crossing Euclidean 3-matching of P, where non-crossing means that no two segments intersect in a common interior point. Both problems, minimum cost and maximum cost non-crossing, are challenging, and we believe that both are NP-hard. Exact solutions to both problems can be attained through integer programming, however, in order to obtain good solutions in feasible times, we fix our attention to heuristics. We present three heuristics specially designed for our problems and compare their solutions and execution times against solving the exact models.

Disclaimer

In this work we present the results of our original research, which have been already published or accepted for publication in [16] and [17]. While this document is structured as a thesis, the text it includes comes from the previous articles in the original language in which they were written (english), with minimal to no edition.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Let P be a set of $n = 3k$ points in the Euclidean plane in general position (i.e., no three points of P are collinear). A 3-matching is a partition of P into k disjoint subsets of 3 points each, called *triplets*. There are several ways of assigning a cost to a triplet, for example the perimeter or the area of the corresponding triangle [3, 13]. In our case, we represent a triplet (u, v, w) by the segments $\overline{uv}$ and $\overline{wv}$ and its cost is given by the sum of the lengths of those segments [3, 8, 10, 14]. The cost of a 3-matching of P is the sum of the costs of its triplets.

In the usual formulation of the Euclidean 3-matching problem we need to find a minimum cost 3-matching of P. See Figure 1.1a. This problem has several applications, especially in the insertion of components on a printed circuit (PC) board [2, 12, 15]. Johnsson, Magyar, and Nevalainen [8, 10] introduced two integer programming formulations for this problem, and proved that its decision version is NP-complete, if each triplet has an arbitrary positive cost (i.e., not necessarily Euclidean). The problem remains NP-complete even if the points of P correspond to vertices of a unit distance graph [16] (a metric cost function).

Besides the minimization problem, we are interested in a similar maximization problem: finding a maximum cost non-crossing Euclidean 3-matching of P. We say that two segments *cross* each other if they intersect in a common interior point. It is easy to see that, in a minimum cost Euclidean 3-matching of P, any two representing segments of triplets do not cross, while in a maximum cost Euclidean 3-matching there could be crossings. See Figure 1.1.

Both problems, minimum cost and maximum cost non-crossing, are challenging, and we believe that both are NP-hard. Similar non-crossing maximization problems have been studied before [1, 4]. Exact solutions to both problems can be attained through Integer Programming, however, in order to obtain good solutions in feasible times, we fix our attention to heuristics.

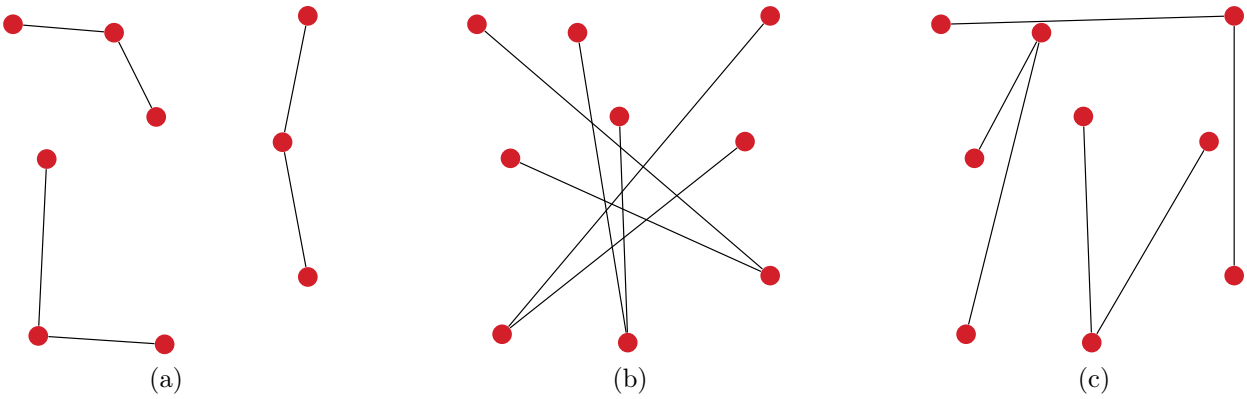


Figure 1.1: Three Euclidean 3-matchings of the same point set P: (a) of minimum cost, (b) of maximum cost, and (c) non-crossing of maximum cost.

This thesis is organized as follows. In Chapter 1 we present the theoretical preliminaries and two well-known integer programming formulations of the minimum Euclidean 3-matching problem. In Chapter 2, we prove that these models have equivalent linear programming relaxations. In Chapter 3, we introduce three new integer programming formulations that use half as many variables and we show how to adapt these formulations to avoid crossings in the maximum non-crossing Euclidean 3-matching problem. We compare the old and new models using some benchmark instances. In Chapter 4 we present three heuristics specially designed for our problems and compare their solutions, using the same benchmark instances as before. Finally, in Chapter 5 we present our conclusions and our ideas for future work.

The remainder of this chapter comes from one of our papers [16].

1.1 Preliminaries

Given two graphs $G = (V, E)$ and $H = (U, F)$, an isomorphism is a bijective function $f : V \to U$ such that $ab \in E$ if and only if $f(a)f(b) \in F$. An H-matching in G is a subgraph M of G in which each connected component is isomorphic to H. If M spans all vertices of G, we say that it is a perfect H-matching. We will assume that H has at least one connected component with at least three vertices, since all problems considered here are polynomially solvable if H does not satisfy this condition [5, 9]. Deciding whether G has a perfect H-matching is NP-complete [9]. In the same way, given an integer $k \geq 0$, it is NP-complete to decide whether G has an H-matching with at least k isomorphic copies of H.

If the edges of G have positive costs $c \in \mathbb{R}^E_+$, the cost of an H-matching M is simply the sum of the costs of the edges in M. Let $b \geq 0$ be a given cost bound. It is NP-complete

to decide whether G has an H-matching with cost at least b. To see this, simply give unit costs to all edges of G and ask whether G has an H-matching of cost at least $\frac{|V|}{|U|}|F|$, which is equivalent to whether G has a perfect H-matching. It is also NP-complete to decide whether G has a perfect H-matching with cost at most b. To see this, again give unit costs to all edges of G and ask whether G has a perfect H-matching of cost at most $\frac{|V|}{|U|}|F|$, which is equivalent to whether G has a perfect H-matching.

If G is a complete graph, then a necessary and sufficient condition for it to have a perfect H-matching is that $|V|$ is a multiple of $|U|$. Even in this case both optimization problems remain NP-complete. However, it is possible to give positive results if c is a metric, that is, for every three vertices $u, v, w \in V$ the triangle inequality $c_{uw} + c_{wv} \geq c_{uv}$ holds. Approximation algorithms for $H \in \{C_3, P_2, P_3\}$ are given in [3, 7, 14].

Our particular interest is the Euclidean metric. In this case, we start with a set of points P in the Euclidean plane. Each of these points corresponds with a vertex of a complete graph $G = (V, E)$, and the cost of each edge is simply the Euclidean distance between the corresponding points. H-matching problems with Euclidean metric have been studied by several authors [3, 11, 13]. We are interested in the special case $H = P_2$, obviously equivalent to the problem described in the introduction.

1.2 State of the Art

In 1998, Johnsson, Magyar, and Nevalainen gave their first integer programming formulation for the problem [8]. Let $G = (V, E)$ be a complete graph, and let $D = (V, A)$ be the directed graph obtained from G by replacing each of its edges $uv \in E$ by the two arcs $uv, vu \in A$. For each arc $uv \in A$, let x_{uv} be a binary variable representing whether arc uv has been chosen ($x_{uv} = 1$) or not ($x_{uv} = 0$) as part of the solution. The main idea is to represent a chosen triplet u, v, w with central vertex v by the two arcs uv, wv. In the following integer program, Equation (1.1) implies that, for each $s \in V$, either the first term is 1 and the second is 0 or viceversa. In the former case, two arcs point towards s, while in the latter case one arc points away from s. In other words, either s is the central vertex of the path or s is an end of the path. See Figure 1.2.

$$\begin{aligned} \min z &= \sum_{rs \in A} c_{rs} x_{rs} \\ \text{subject to} & \\ \frac{1}{2} \sum_{rs \in A} x_{rs} + \sum_{sk \in A} x_{sk} &= 1 \; \forall s \in V \qquad (1.1) \\ x_{rs} &\in \{0,1\} \; \forall rs \in A. \qquad (1.2) \end{aligned}$$

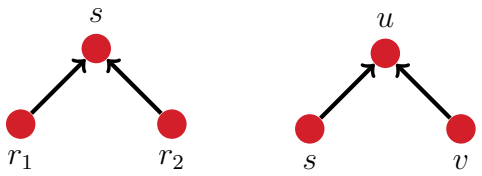


Figure 1.2: The two possibilities for s in the first model.

In 1999, Johnsson, Magyar, and Nevalainen gave their second integer programming formulation for the problem [10]. Let $G = (V, E)$ be a complete graph, and let $D = (V, A)$ be the directed graph obtained from G by replacing each of its edges $uv \in E$ by the two arcs $uv, vu \in A$, and for each vertex $v \in V$ adding a loop from v to v. For each arc $uv \in A$, let y_{uv} be a binary variable representing whether arc uv has been chosen ($y_{uv} = 1$) or not ($y_{uv} = 0$) as part of the solution. The main idea is to represent a chosen triplet u, v, w with central vertex v by the three arcs vv, vu, vw.

$$\begin{aligned} \min z &= \sum_{rs \in A, r \neq s} c_{rs} y_{rs} \\ \text{subject to} & \\ \sum_{rs \in A} y_{rs} &= 1 \; \forall s \in V \qquad (1.3) \\ \sum_{s \in V} y_{ss} &= \frac{1}{3}|V| \qquad (1.4) \\ \sum_{sr \in A, r \neq s} y_{sr} &= 2y_{ss} \; \forall s \in V \qquad (1.5) \\ y_{rs} &\in \{0,1\} \; \forall rs \in A. \qquad (1.6) \end{aligned}$$

Equation (1.3) implies that exactly one arc enters each vertex. Equation (1.4) implies

that exactly one third of the vertices are chosen as centers. Equation (1.5) implies that if s is a center, then exactly two arcs leave s. See Figure 1.3.

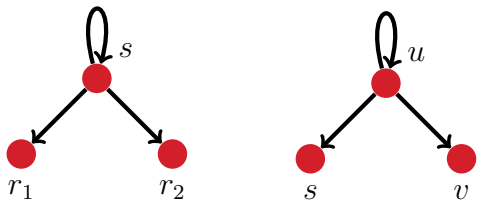


Figure 1.3: The two possibilities for s in the second model.

Although they did not use their formulations in a more general setting, observe that G could have been chosen as a general undirected graph and the costs are not forced to be Euclidean (or even metric). Furthermore, we could have even chosen an asymmetric D with asymmetric costs.

These models were tested against a benchmark available at `http://www.cs.utu.fi/research/projects/3mp/`. All instances in that benchmark consist of point sets in the Euclidean plane. At the time, Johnsson, Magyar, and Nevalainen reported the optimal solutions to most instances with 99 vertices or less [8, 10]. See Figure 1.4. In Table 1.1 we report the optimal values of all instances with 120 vertices or less and the value of the linear programming relaxation of their models and the *integrality gap*, given as the percentage difference between the optimal value and the relaxation value. As observed before [10], the value of the relaxations of both models coincide for all instances in the benchmark. Figures 1.5 and 1.6 show, for the same instance, the fractional solutions for both linear programming relaxations (edges are colored black if chosen, white if not chosen, and in various shades of gray if chosen fractionally).

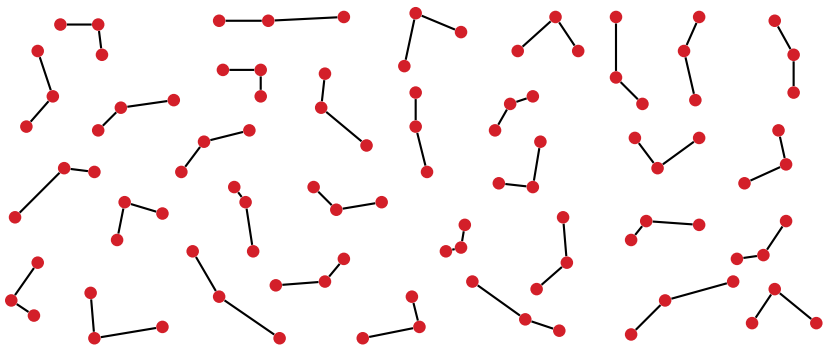


Figure 1.4: Optimal solution for instance h03.

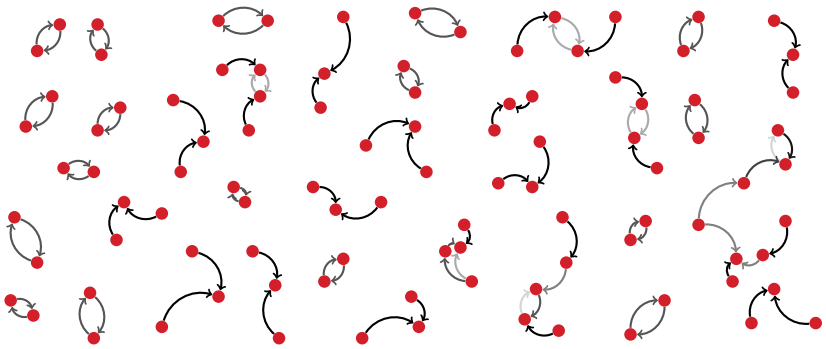


Figure 1.5: Relaxation for instance h03 in the 1998 model.

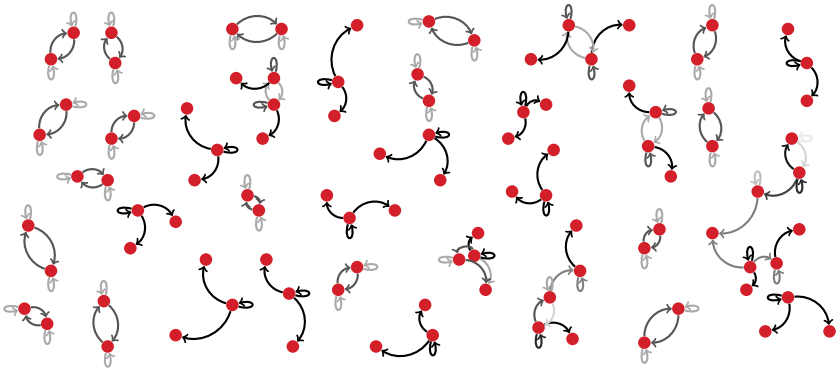


Figure 1.6: Relaxation for instance h03 in the 1999 model.

Instance	Number of vertices	Optimal value	Relaxation value	Integrality gap
f21	21	159.7289	146.4494	08.31%
f27	27	8011.0787	7379.4580	07.88%
f33	33	255.6944	236.9880	07.32%
f39	39	282.3146	237.7430	15.79%
39a	39	783.9551	705.7473	09.98%
39b	39	826.1430	685.4586	17.03%
39c	39	959.3756	865.2224	09.81%
39d	39	781.2205	699.9003	10.41%
39e	39	872.3539	758.9849	13.00%
42a	42	949.0908	805.3827	15.14%
42b	42	860.5917	727.7668	15.43%
45a	45	1013.1434	902.5522	10.92%
45b	45	985.6019	785.1058	20.34%
48a	48	996.6131	945.5751	05.12%
48b	48	967.8344	817.5964	15.52%
51a	51	983.5503	843.9761	14.19%
51b	51	1003.8185	892.6889	11.07%
h01	51	**265.6100**	**243.4752**	08.33%
h02	84	**1007.1086**	**876.5153**	12.97%
man	84	**1007.1086**	**876.5153**	12.97%
h03	99	**751.5259**	**684.7681**	08.88%
f99	99	386.2317	363.1123	05.99%
rat99	99	751.5259	684.7681	08.88%
120a	120	**1508.7162**	**1229.2390**	18.52%

Table 1.1: Benchmarks, optimal values, relaxation values, and integrality gaps for the original integer programming formulations. The values in **bold** correspond to instances not solved in [8, 10].

Chapter 2

Equivalence of the Linear Relaxations

In this chapter, we prove that the models from the state of the art have equivalent linear relaxations. The authors of these models had already noticed that the relaxation values of their models were approximately the same for a set of instances, but no formal proof of equivalence was offered. The slight differences noticed were due to their choice of Mixed Integer Programming (MIP) solver. We prove that the relaxation values coincide for any given instance. The content of this chapter comes from one of our papers [16].

2.1 Mathematical proof

We prove now that both linear programming relaxations are equivalent. In particular, we construct feasible solutions to both systems with the same cost.

Let $\mathbf{0} \leq y \leq \mathbf{1}$ be a feasible solution to the linear system (1.3–1.5). Let x be the vector given by $x_{rs} = y_{sr}$ for all $r, s \in V$ with $r \neq s$. Clearly $\mathbf{0} \leq x \leq \mathbf{1}$. For each $s \in V$ we have

$$\begin{aligned} \tfrac{1}{2} \sum_{rs \in A} x_{rs} + \sum_{sk \in A} x_{sk} &= \tfrac{1}{2} \sum_{sr \in A, r \neq s} y_{sr} + \sum_{ks \in A} y_{ks} && (2.1) \\ &= \tfrac{1}{2}(2y_{ss}) + (1 - y_{ss}) = 1, && (2.2) \end{aligned}$$

that is, x satisfies the linear system (1.1).

Conversely, let $\mathbf{0} \leq x \leq \mathbf{1}$ be a feasible solution to the linear system (1.1). Let y be the vector given by $y_{rs} = x_{sr}$ for all $r, s \in V$ with $r \neq s$, and $y_{ss} = 1 - \sum_{rs \in A, r \neq s} y_{rs} \leq 1$ for all

$s \in V$. Also, for each $s \in V$ we have

$$\begin{aligned} y_{ss} &= 1 - \sum_{rs \in A, r \neq s} y_{rs} && (2.3)\\ &= 1 - \sum_{sk \in A} x_{sk} && (2.4)\\ &= \frac{1}{2} \sum_{rs \in A} x_{rs} \geq 0, && (2.5) \end{aligned}$$

and hence $\mathbf{0} \leq y \leq \mathbf{1}$. By definition, (1.3) holds. For $s \in V$

$$\begin{aligned} \sum_{sr \in A, r \neq s} y_{sr} &= \sum_{rs \in A} x_{rs} && (2.6)\\ &= 2\left(1 - \sum_{sr \in A} x_{sr}\right) && (2.7)\\ &= 2\left(1 - \sum_{rs \in A, r \neq s} y_{rs}\right) && (2.8)\\ &= 2y_{ss}, && (2.9) \end{aligned}$$

and hence (1.5) holds. Finally, (1.3) and (1.5) imply (1.4). Adding y_{ss} to both sides of (1.5) and summing over $s \in V$ we obtain

$$\begin{aligned} 3\sum_{s \in V} y_{ss} &= \sum_{s \in V} \sum_{sr \in A} y_{sr} && (2.10)\\ &= \sum_{r \in V} \sum_{rs \in A} y_{rs} && (2.11)\\ &= \sum_{r \in V} 1 = |V|. && (2.12) \end{aligned}$$

In other words, the constraints (1.4) are implied by the constraints (1.3) and (1.5) and therefore are unnecessary.

Chapter 3

New Integer Programming Models

In this chapter we present three new integer programming formulations that use half as many variables than those of Johnsson, Magyar, and Nevalainen [8, 10] and we show how to adapt these formulations to avoid crossings in the maximum non-crossing Euclidean 3-matching problem. Using both the old and new models, we solve to optimality some benchmark instances and we also compare their linear programming relaxations. The content of this chapter comes from one of our papers [17].

3.1 Triplet integer programming formulation

The two models by Johnsson, Magyar, and Nevalainen [8, 10] have the possible disadvantage of requiring two variables for each pair of points of P, whereas the following *triplet* model has only one variable for each pair of points (and no variables associated to the central points of a triplet). Let $U = (P, E)$ be the geometric complete graph, so that for every two different points $u, v \in P$ we have the line segment $\overline{uv} \in E$. For each $S \subseteq P$, let $\delta(S)$ be the set of edges with one end point in S and the other in $P \setminus S$ and let $\gamma(S)$ be the set of edges with both end points in S. For each edge $\overline{uv} \in E$, let x_{uv} be a binary variable representing whether such edge has been chosen ($x_{uv} = 1$) or not ($x_{uv} = 0$). For each $F \subset E$, let $x(F) = \sum_{\overline{uv} \in F} x_{uv}$.

Consider now the integer program [16]:

$$\begin{aligned}
\min z &= \sum_{\overline{rs}\in E} c_{rs}x_{rs} \\
\text{subject to} & \\
x(\delta(v)) &\geq 1 \; \forall v \in P && (3.1)\\
x(\delta(r,s,t)) &\leq 3(2 - x(\gamma(r,s,t))) \; \forall r,s,t \in P && (3.2)\\
x(\delta(r,s,t)) &\geq \frac{3}{2}(2 - x(\gamma(r,s,t))) \; \forall r,s,t \in P && (3.3)\\
x_{rs} &\in \{0,1\} \; \forall \overline{rs} \in E. && (3.4)
\end{aligned}$$

Constraint (3.1) implies that each point must be incident to at least one chosen edge. Now consider the point set $\{r,s,t\}$. If a triplet of this point set is chosen, then $x(\gamma(r,s,t)) = 2$ and $x(\delta(r,s,t)) = 0$. If instead only one edge in $\gamma(r,s,t)$ is chosen, then $x(\gamma(r,s,t)) = 1$ and $2 \leq x(\delta(r,s,t)) \leq 3$. Finally, if no edge in $\gamma(r,s,t)$ is chosen, then $x(\gamma(r,s,t)) = 0$ and $3 \leq x(\delta(r,s,t)) \leq 6$. Note that all this cases are valid for constraints (3.2) and (3.3). See Figure. 3.1.

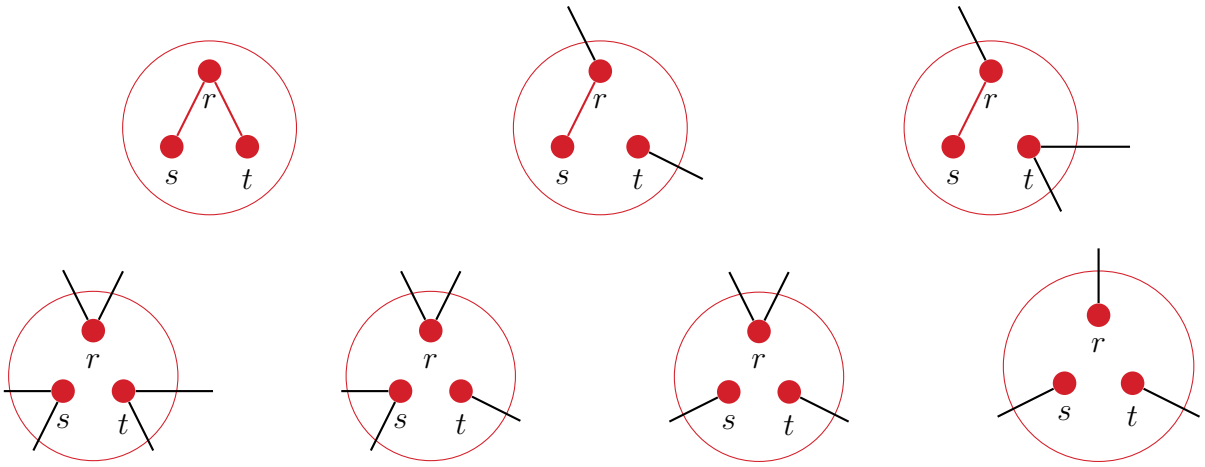


Figure 3.1: All valid cases for our model. The edges from $\gamma(r,s,t)$ are shown in red.

Conversely, the three disallowed cases are: paths with more than two edges, points with more than two edges, and isolated edges. First, let r,s,t,u be consecutive points in a path of three (or more) edges. If $u = r$, then we have $x(\gamma(r,s,t)) = 3$, violating (3.2). If $u \neq r$, then $x(\gamma(r,s,t)) = 2$ and $x(\delta(r,s,t)) \geq 1$, also violating (3.2). Second, let r be a point with at least three neighbors s,t,u. Then $x(\gamma(r,s,t)) \geq 2$ and $x(\delta(r,s,t)) \geq 1$, also violating (3.2). Finally, let $\overline{rs}$ be an isolated edge and let $t \in P \setminus \{r,s\}$. Constraints (3.2) and (3.3) imply that t has degree either 2 or 3. Since the latter is impossible, it follows that t has degree 2

and, therefore, we have selected a set of cycles. This is a contradiction, because paths with three or more edges are impossible. See Figure. 3.2.

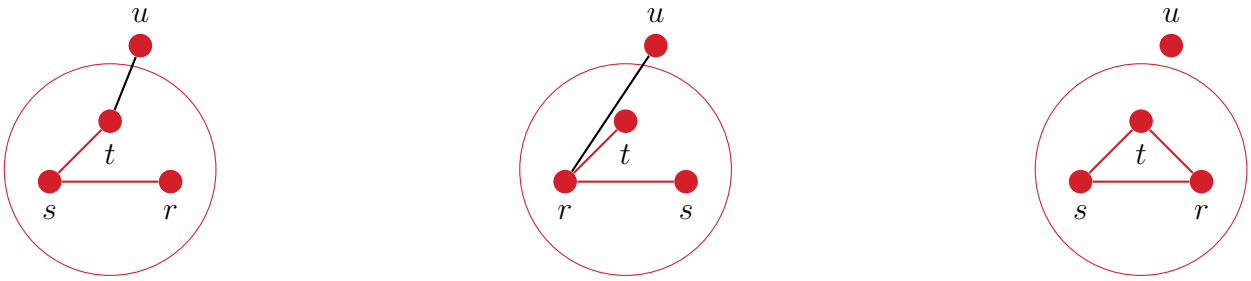


Figure 3.2: Invalid cases for our model.

3.2 Pair and Quad integer programming formulations

Here we introduce two additional models for the minimum cost problem. We call them *Pair integer programming formulation* and *Quad integer programming formulation*, due to the maximum number of points involved in each of their constraints. Because of their similarities, both models are described together.

We take the geometric graph $U = (P, E)$, and the functions $\delta(S)$, $\gamma(S)$ and $x(F)$, as they were defined in Section. 3.1.

The Pair integer program is defined as:

$$\begin{aligned}
\min z &= \sum_{\overline{rs} \in E} c_{rs} x_{rs} \\
\text{subject to} & \\
x(\delta(v)) &\geq 1 \; \forall v \in P && (3.5)\\
x(\delta(r)) + x(\delta(s)) &\geq 2 + x_{rs} \; \forall \overline{rs} \in E && (3.6)\\
x(\delta(r)) + x(\delta(s)) &\leq 4 - x_{rs} \; , \forall \overline{rs} \in E && (3.7)\\
x_{rs} &\in \{0, 1\} \; \forall \overline{rs} \in E && (3.8)
\end{aligned}$$

The Quad integer program is defined as:

$$\begin{aligned} \min z &= \sum_{\overline{rs} \in E} c_{rs} x_{rs} \\ \text{subject to} & \\ x(\delta(v)) &\geq 1 \; \forall v \in P && (3.9) \\ x(\delta(r,s)) &\geq 1 \; \forall \overline{rs} \in E && (3.10) \\ x(\gamma(r,s,u,v)) &\leq 2 \; \forall (r,s,u,v) \in P && (3.11) \\ x_{rs} &\in \{0,1\} \; \forall \overline{rs} \in E && (3.12) \end{aligned}$$

Constraints (3.5) and (3.9) imply that each point must be incident to at least one chosen edge. Now consider the point set $\{r, s, t\}$, and suppose that a triplet of this point set is chosen. We have two cases, either r is the central point of the triplet or it is not. In both cases $2+1 \leq x(\delta(r)) + x(\delta(s)) \leq 4-1$ and $x(\gamma(r,s,u,v)) \leq 2$; those values are valid for the constraints of the two models. See Figure. 3.3.

(a) (b)

Figure 3.3: Valid cases for both models. The segment $\overline{rs}$ is shown in blue.

Suppose now that some segment, $\overline{rs}$, was not chosen. As r and s must belong to some triplet, we can only have the cases depicted in Figure 3.4. In all cases $2+0 \leq x(\delta(r)) + x(\delta(s)) \leq 4-0$ and $x(\delta(r,s)) \geq 2$; those values are also valid for the constraints of the two models.

Conversely, the disallowed cases are: paths with more than two edges, points with more than two edges, triangles and isolated edges. In this case $x(\delta(r)) + x(\delta(s)) > 4 - x_{rs}$, violating (3.7). If $u \neq v$, then $x(\gamma(r,s,u,v)) > 2$, violating (3.11). Finally, if $u = v$, let w be any vertex not in $\{u, r, s, v\}$. Then $x(\gamma(r,s,u,w)) > 2$, also violating (3.11). Therefore, triangles and paths with more than two edges are disallowed. Second, let r be the center of a star with neighbors u, v, and s. In this case $x(\delta(r)) + x(\delta(s)) > 4 - x_{rs}$, violating (3.7), and $x(\gamma(r,s,u,v)) > 2$, violating (3.11). Therefore, points with more than two edges

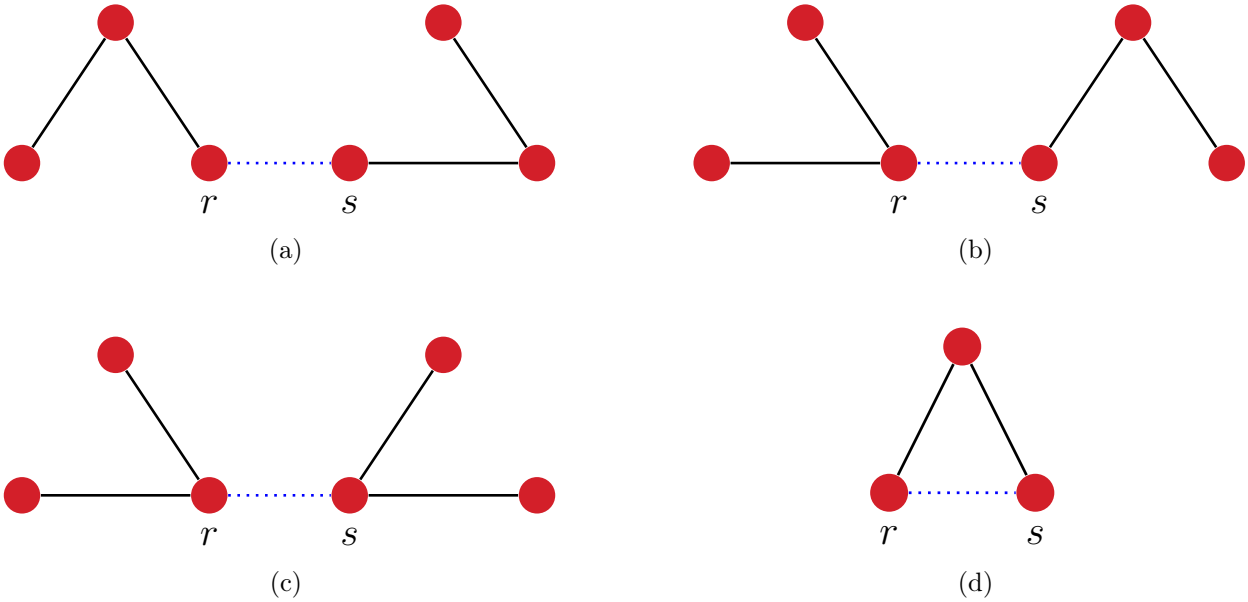


Figure 3.4: More valid cases for both models.

are disallowed. The same inequalities hold for the case when r is touched by two edges (or more). Finally, constraints (3.6) and (3.10) exclude isolated edges. See Figure. 3.5.

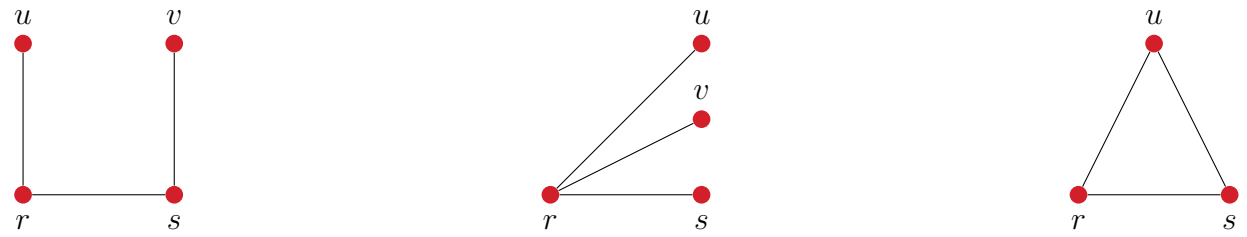


Figure 3.5: Invalid cases for Pair and Quad models.

3.3 Experimental results

Additionally, using the state-of-the-art MIP solver Gurobi [6], we solved to optimality all instances that were used as tests in [8, 10], in the benchmark available at `http://www.cs.utu.fi/research/projects/3mp/`. Each instance in that benchmark is a point set in the Euclidean plane, of size divisible by 3.

Table 3.1 contains, for each instance, the relaxation value and integrality gap of all previous models. The integrality gap is taken as the ratio between the optimal value and the

relaxation value. Naturally, the linear relaxation values differ between models.

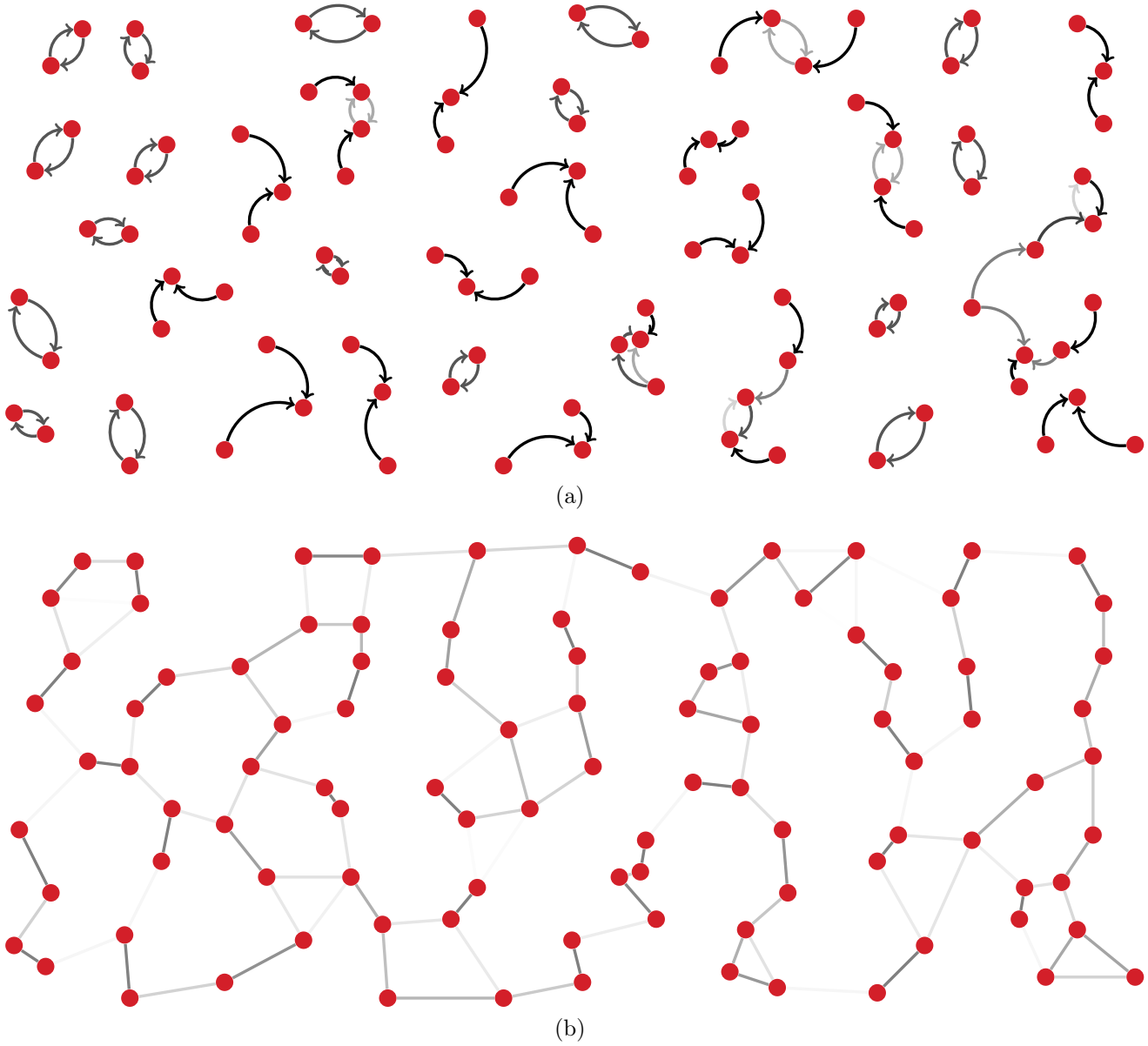
Table 3.1: Relaxation values and integrality gaps of all models. Tighter relaxations are shown in bold.

Case	n	Minimum value	Relaxation 1998/1999	Integrality gap	Relaxation Pair	Integrality gap	Relaxation Triplet	Integrality gap	Relaxation Quad	Integrality gap
f21	21	159.7289	146.4494	1.0907	156.5128	1.0205	141.2287	1.1310	156.6416	**1.0197**
f27	27	8011.0786	7379.4580	1.0856	7791.3999	1.0282	7182.3311	1.1154	7837.2612	**1.0222**
f33	33	255.6944	236.9880	1.0789	255.6944	**1.0000**	232.2087	1.1011	255.6944	**1.0000**
f39	39	282.3146	237.7430	1.1875	251.3423	1.1232	237.4670	1.1889	257.4570	**1.0966**
39a	39	783.9551	705.7473	1.1108	769.1493	1.0192	718.6500	1.0909	775.6470	**1.0107**
39b	39	826.1430	685.4586	1.2052	764.0848	1.0812	680.2161	1.2145	781.6820	**1.0569**
39c	39	959.3756	865.2224	1.1088	944.3328	1.0159	866.2079	1.1076	946.0240	**1.0141**
39d	39	781.2205	699.9003	1.1162	766.1649	1.0197	709.9151	1.1004	781.2205	**1.0000**
39e	39	872.3539	758.9849	1.1494	813.9905	1.0717	739.8327	1.1791	831.5713	**1.0490**
42a	42	949.0908	805.3827	1.1784	882.7597	1.0751	810.1607	1.1715	903.2889	**1.0507**
42b	42	860.5917	727.7668	1.1825	806.2082	1.0675	735.9717	1.1693	814.2614	**1.0569**
45a	45	1013.1434	902.5522	1.1225	1003.4940	1.0096	905.4010	1.1190	1013.1430	**1.0000**
45b	45	985.6019	785.1058	1.2554	940.5861	1.0479	810.2633	1.2164	962.0711	**1.0245**
48a	48	996.6130	945.5751	1.0540	996.3883	1.0002	936.0416	1.0647	996.6131	**1.0000**
48b	48	967.8344	817.5964	1.1838	961.9909	1.0061	832.4138	1.1627	967.8344	**1.0000**
51a	51	983.5503	843.9761	1.1654	962.6867	1.0217	850.3052	1.1567	979.7631	**1.0039**
51b	51	1003.8185	892.6889	1.1245	971.0931	1.0337	883.1294	1.1367	994.0304	**1.0098**
h01	51	265.6100	243.4752	1.0909	255.2249	1.0407	232.7736	1.1411	255.8900	**1.0380**
h02	84	1007.1086	876.5153	1.1490	947.6500	1.0627	870.1146	1.1574	964.0787	**1.0446**
man	84	1007.1086	876.5153	1.1490	947.6500	1.0627	870.1146	1.1574	964.0787	**1.0446**
h03	99	751.5259	684.7681	1.0975	736.4505	1.0205	666.2244	1.1280	737.9276	**1.0184**
f99	99	386.2318	363.1123	1.0637	377.9442	1.0219	351.9001	1.0976	379.3111	**1.0182**
rat99	99	751.5259	684.7681	1.0975	736.4505	1.0205	666.2244	1.1280	737.9276	**1.0184**
120a	120	1508.7162	1229.2390	1.2274	1430.6870	1.0545	1251.5170	1.2055	1466.5570	**1.0287**

In our experiments the Quad model had better relaxation than the rest of the models, we conjecture that this is always the case.

Another observation comes from drawing the fractional solutions of the linear programming relaxations (edges are colored black if chosen, white if not chosen, and in various shades of gray if chosen fractionally). As the two models by Johnsson, Magyar, and Nevalainen [8, 10] have equivalent linear programming relaxations, the drawings of the solutions look the same for each instance, as expected. However, it seems that there are no edges with positive value that cross (as in Figure 3.6a). Since this happened for each experiment, we also conjecture that this is always the case. The same property was seen in each experiment of the Triplet model (as in Figure 3.6b), again we conjecture that this is always the case. See Figure. 3.6.

Conversely, we know that such property is not true for the Pair model, since instance 39b has at least one crossing. See Figure. 3.7.



(a)

(b)

Figure 3.6: Relaxation for instance h03: (a) in the 1998 model and (b) in the Triplet model.

3.4 Models for the maximum cost problem

Simply changing the objective function from minimization to maximization does not work: The resulting maximum cost 3-matching will almost certainly contain many crosses (as in Figure 1.1b). Therefore, in order to obtain an integer programming formulation for the maximum cost non-crossing 3-matching problem, we need to add a family of constraints to

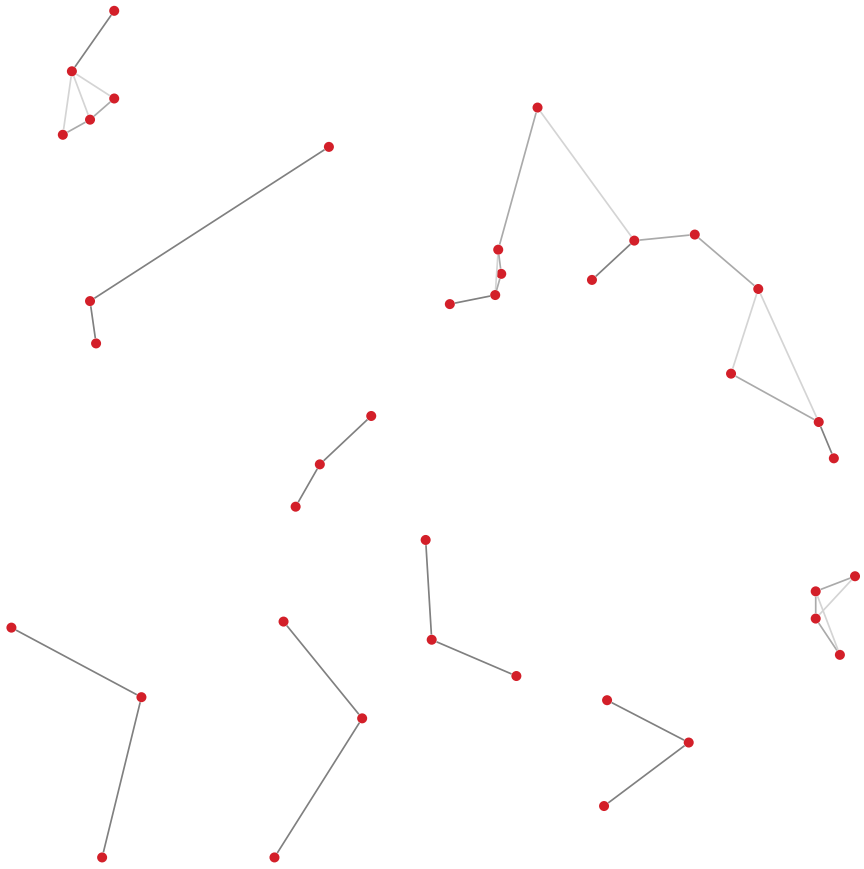


Figure 3.7: Relaxation for instance 39b in the Pair model. The crossing appears in the lower right corner.

deal with crossings.

Note that a crossing between $\overline{ik}$ and $\overline{jl}$ occurs if and only if i, j, k, l are the vertices of a convex quadrangle ($\overline{ik}$ and $\overline{jl}$ would be the diagonals of this quadrangle). Hence, we can add these constraints to the first two models, with $i, j, k, l \in P$:

$$x_{ik} + x_{ki} + x_{jl} + x_{lj} \leq 1 \quad \text{if } i, j, k, \text{ and } l \text{ form a convex quadrangle.} \tag{3.13}$$

Or we can add these other constraints to the last three models, with $i, j, k, l \in P$:

$$x_{ik} + x_{jl} \leq 1 \quad \text{if } i, j, k, \text{ and } l \text{ form a convex quadrangle.} \tag{3.14}$$

In either case, these $O(n^4)$ constraints imply that we can only select one of the two diagonals of each convex quadrangle, therefore avoiding crossings.

Chapter 4

Three Geometric Heuristics

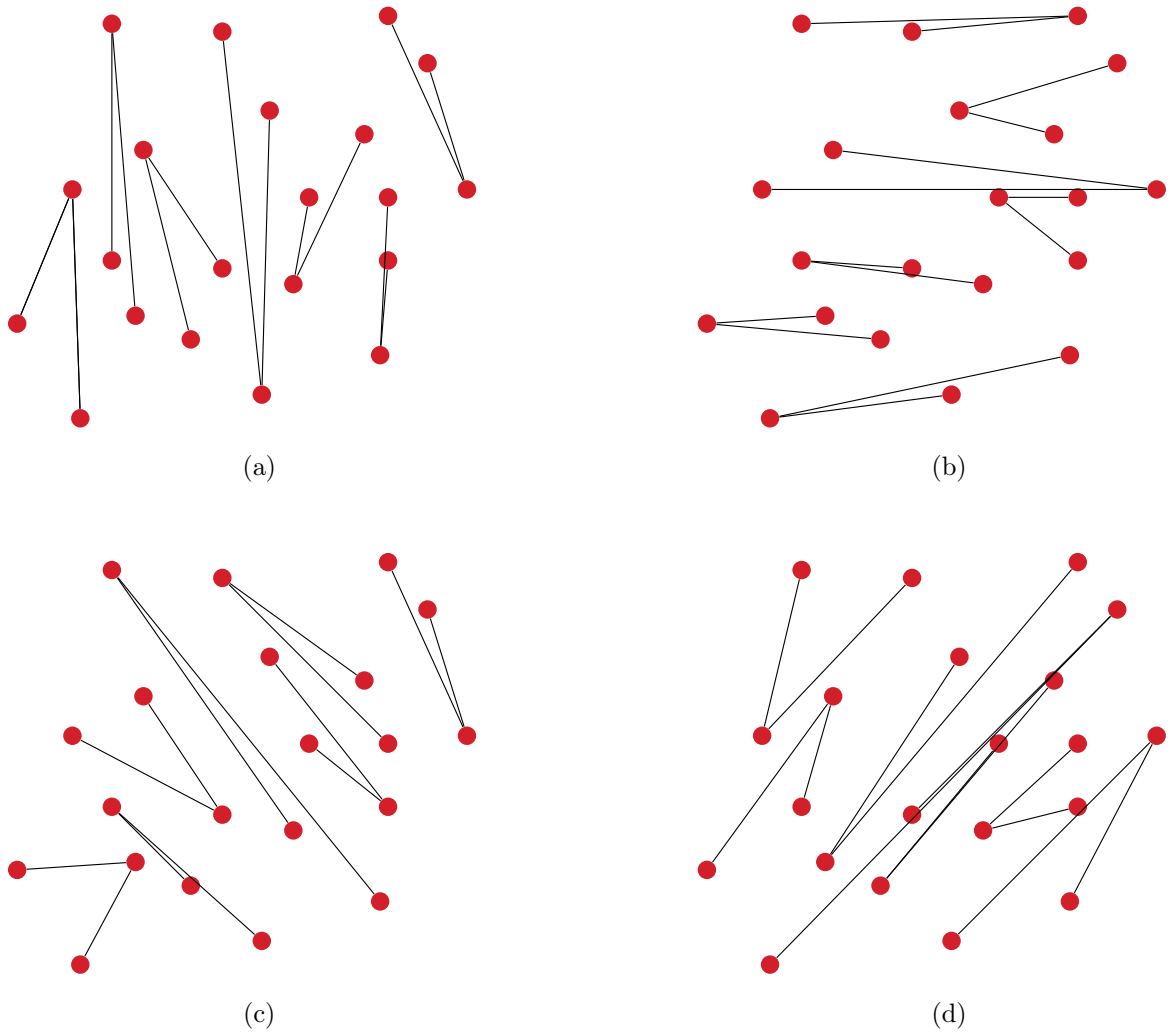
In this chapter we introduce three heuristics specially designed to compute 3-matchings of a set of points. All three are used for both: minimization and maximization problems; with slight modifications. We use the benchmark mentioned in Chapter 3 to test these strategies. The content of this chapter comes from one of our papers [17].

4.1 Statements of the heuristics

The following statements are for the minimization problem. For the maximization problem, the necessary modifications are explained in parenthesis.

4.1.1 Windrose

Sort the points of P along the x-axis and then partition P in subsets of 3 consecutive points. For each of these subsets consider the permutation of its elements (u, v, w) that minimizes (resp. maximizes) the sum of the lengths of $\overline{uv}$ and $\overline{wv}$, and add this triplet to the solution of this direction. Repeat this process along the y-axis, along a $45°$ line and along a $-45°$ line. Among the four solutions, select that of minimum cost (resp. maximum cost). See Figure. 4.1.



(a) (b) (c) (d)

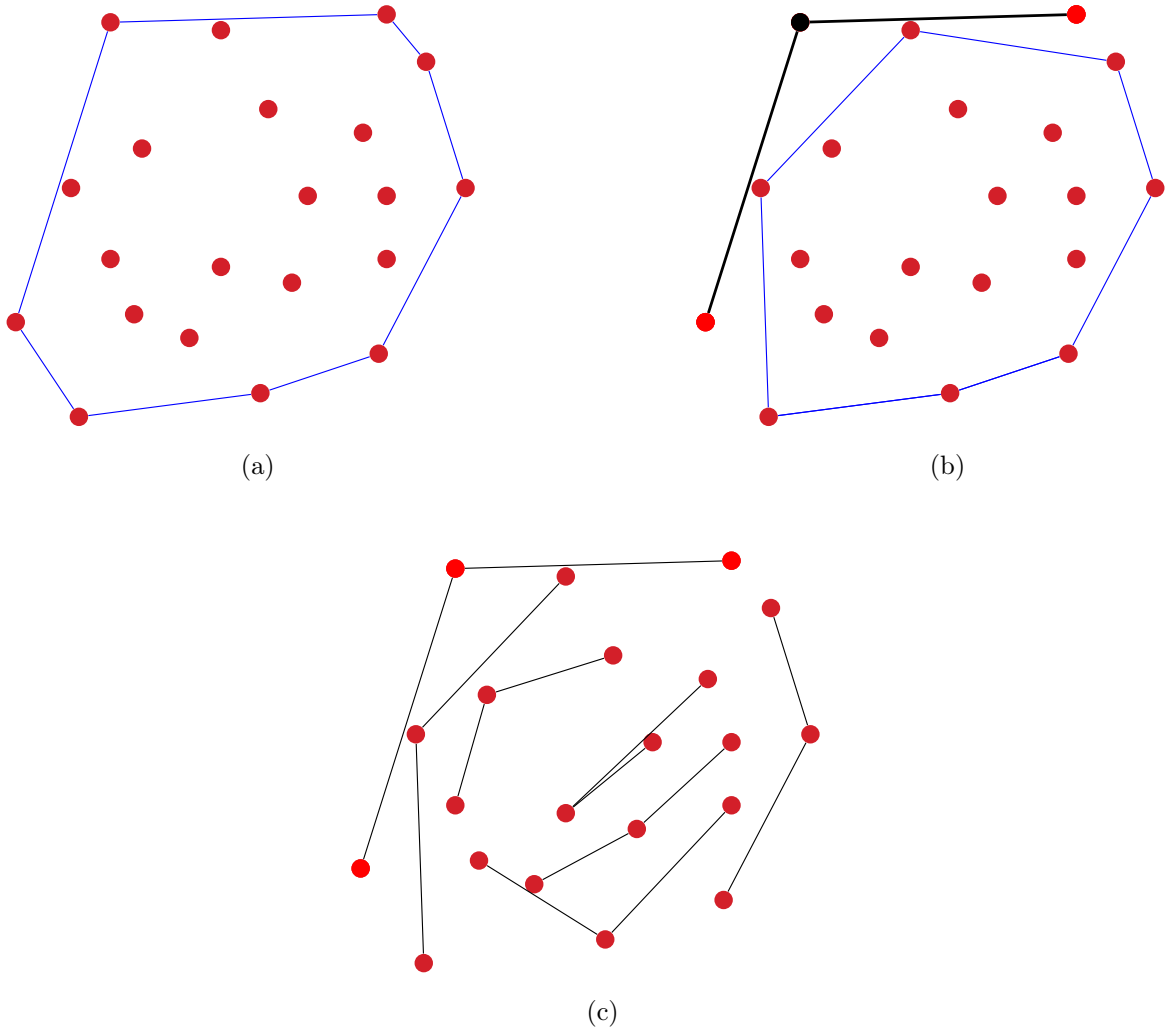
Figure 4.1: The four stages of (high cost) Windrose heuristic for instance f21: (a) along the x-axis, (b) along the y-axis, (c) along a 45° line, and (d) along a −45° line.

Algorithm 4.1 Windrose Heuristic

function WINDROSE($P \subseteq \mathbb{R} \times \mathbb{R}, \lambda \in \{\min, \max\}$)
  $R \leftarrow \emptyset, c_R \leftarrow 0$
  for all $d \in \{0°, 45°, -45°, 90°\}$ **do**
    $P' \leftarrow sort_d(P)$
    $T \leftarrow \{optimal(\{P'_{3i+1}, P'_{3i+2}, P'_{3i+3}\}, \lambda) : 0 \leq i < \frac{|P|}{3}\}$
    $c_T = \sum_{i=1}^{\frac{|P|}{3}} (d(T_{i,1}, T_{i,2}) + d(T_{i,2}, T_{i,3}))$
    if $R = \emptyset \vee \lambda(c_R, c_T) = c_T$ **then**
      $R \leftarrow T, c_R \leftarrow c_T$
  return R

4.1.2 ConvHull

At each step: compute conv(P), the convex hull of P (see Figure. 4.2a); search for two consecutive segments in conv(P) whose sum of lengths is minimum (resp. maximum); add the induced triplet of points to the solution and delete them from P. Repeat this process (Fig. 4.2b) until P is empty. In each step we remove points from the current conv(P), so all the segments contained in the final solution do not cross each other. See Figure. 4.2.



(a) (b) (c)

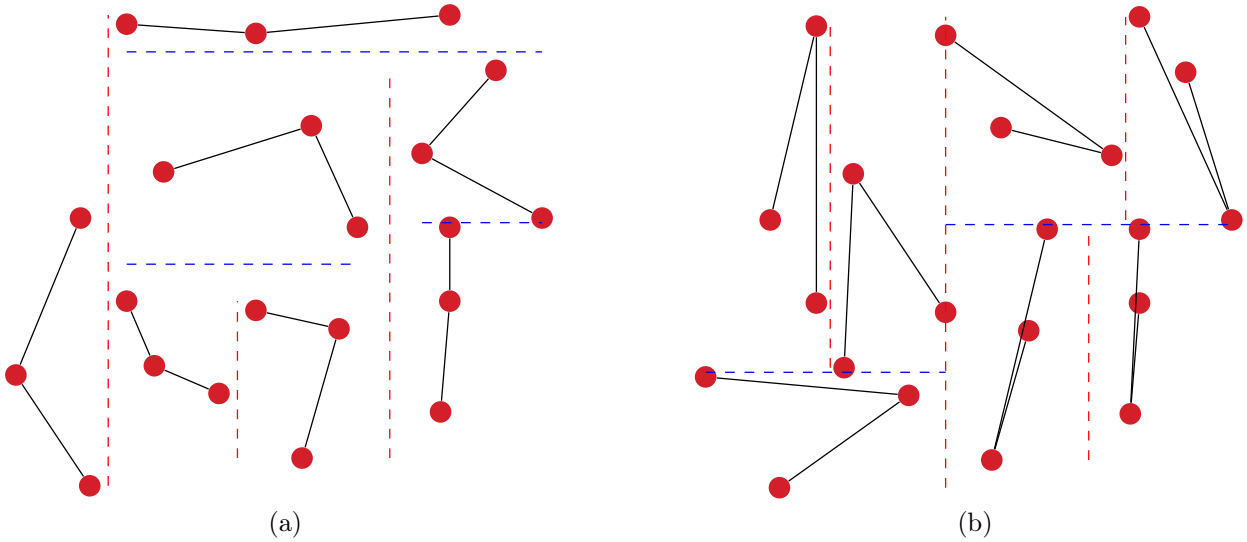
Figure 4.2: The (high cost) ConvHull heuristic applied to instance f21: (a)–(b) first pass of the heuristic, and (c) the resulting solution.

Algorithm 4.2 ConvHull Heuristic

function ConvHull($P \subseteq \mathbb{R} \times \mathbb{R}, \lambda \in \{\min, \max\}$)
  if $|P| = 0$ **then return** $\emptyset$
  $P' \leftarrow sort_{clockwise}(conv(P))$
  $r \leftarrow \{min \mapsto <\ ;\ max \mapsto >\}$
  $k \leftarrow i : r(\lambda)(d(P'_i, P'_{i+1}) + d(P'_{i+1}, P'_{i+2}), d(P'_j, P'_{j+1}) + d(P'_{j+1}, P'_{j+2})), 1 \leq i \neq j \leq |P| - 2$
  $T \leftarrow \{P'_k, P'_{k+1}, P'_{k+2}\}$
  return $optimal(T, \lambda) \cup ConvHull(P \setminus T, \lambda)$

4.1.3 Guillotine

Given a point p, denote by $x(p)$ its x-coordinate and by $y(p)$ its y-coordinate. The set P has $n = 3k$ points. If $n = 3$ consider the triplet (u, v, w) that minimizes (resp. maximizes) the cost and add it to the solution. If $n > 3$ and the x-axis is selected: Sort the points in P along the x-axis and label them accordingly as $p_1, p_2, \ldots, p_n$. Search for the $i \in \{1, \ldots, k-1\}$ such that $|x(p_{3i}) - x(p_{3i+1})|$ is maximum (resp. minimum). Take $S = \{p_1, \ldots, p_{3i}\}$ and $T = \{p_{3i+1}, \ldots, p_n\}$, and proceed recursively in S and T selecting now the y-axis. The case for the y-axis is analogous. See Figure. 4.3.



(a) (b)

Figure 4.3: Guillotine heuristic applied to instance f21: (a) low cost and (b) high cost.

Algorithm 4.3 Guillotine Heuristic

function GUILLOTINE($P \subseteq \mathbb{R} \times \mathbb{R}, \lambda \in \{\min, \max\}, s \in \{1, 2\}$)
  if $|P| = 3$ **then return** *optimal*(P, λ)
  $P' \leftarrow sort_{x_s}(P)$
  $r \leftarrow \{min \mapsto >\ ;\ max \mapsto <\}$
  $k \leftarrow i \in \mathbb{N} : r(\lambda)(|x_d(P'_{3i}) - x_d(P'_{3i+1})|, |x_s(P'_{3j}) - x_s(P'_{3j+1}|), 1 \leq i \neq j < \frac{|P|}{3}$
  $Q_1 \leftarrow p \in P : x_s(p) \leq x_s(P'_{3k})$
  $Q_2 \leftarrow p \in P : x_s(P'_{3k+1}) \leq x_s(p)$
  $s' \leftarrow (s \bmod 2) + 1$
  return $Guillotine(Q_1, \lambda, s') \cup Guillotine(Q_2, \lambda, s')$

4.2 Experimental results

Tables 4.1, 4.2 and 4.3 summarize the results obtained by our heuristics, compared against the optimal values. The optimal values were computed with the MIP solver Gurobi [6]. For the minimum cost 3-matching of P we used the first integer model of Johnsson, Magyar, and Nevalainen, and for the maximum cost we adapted that formulation as described in the Section 3.4.

4.2.1 Low cost 3-matchings

In Table 4.1 we see first that it is feasible to obtain the corresponding minimum value using Gurobi even with sets of about 200 points, usually in less than 30 seconds. Then we report the results obtained in 2000 by Johnsson, Magyar, and Nevalainen [11] using several heuristic strategies, primarily genetic algorithms. Their results are very near to the optimum, usually within 1% of the minimum. However, they needed about one hour of time in a Pentium 133 to obtain them. Even considering that our hardware is about 50 times faster, their heuristic needed more time to obtain an approximate solution than Gurobi needed to obtain the optimal solution. The rest of the columns of Table 4.1 and Table 4.2 show the results obtained by our heuristics. All our heuristics are very fast (the whole set of tests ran in only a few seconds), but only the Guillotine heuristic gave reasonably good solutions. It is likely that this heuristic is a 2-approximation algorithm for the minimization problem. This intuition comes from reading the ratio column, that is, the quotient cost of the solution obtained by the heuristic divided by the cost of the optimal solution. Clearly such quotient is ≥ 1 for a minimization problem, and the closer it is to 1, the better.

Table 4.1: Comparison of the average best results in [11] (among 20 runs) and our heuristics, for the minimum cost problem.

Case	n	minimum	$\text{time}_{[s]}$	[11]	Ratio	Windrose	Ratio	ConvHull	Ratio	Guillotine	Ratio
p01	201	1862.29	10.74	1869.63	1.00	9812.23	5.27	4715.08	2.53	2318.07	1.24
p02	201	1826.01	21.01	1835.51	1.01	9662.64	5.29	4871.67	2.67	2540.78	1.39
p03	201	1812.18	7.20	1815.13	1.00	9040.09	4.99	4629.54	2.55	2404.50	1.33
p04	201	1839.27	28.50	1844.78	1.00	8621.45	4.69	4214.26	2.29	2513.79	1.37
p05	201	1856.08	8.62	1872.04	1.01	9611.48	5.18	4999.93	2.69	2598.20	1.40
p06	201	1775.64	19.49	1786.95	1.01	9048.62	5.10	4838.06	2.72	2333.94	1.31
p07	201	1802.04	10.13	1816.51	1.01	8880.21	4.93	4907.46	2.72	2392.24	1.33
p08	201	1822.25	36.71	1838.47	1.01	8794.89	4.83	4904.52	2.69	2100.94	1.15
p09	201	1846.29	19.91	1851.40	1.00	9459.49	5.12	5171.86	2.80	2506.06	1.36
p10	201	1913.10	8.20	1919.95	1.00	9194.90	4.81	5342.14	2.79	2653.62	1.39
p11	150	15227.38	5.32	15228.28	1.00	51179.90	3.36	36207.00	2.38	19487.60	1.28
p12	150	3908.57	4.49	3922.06	1.00	15401.40	3.94	9430.38	2.41	5310.42	1.36
p13	195	1438.88	6.61	1447.40	1.01	4581.78	3.18	2885.85	2.01	1972.89	1.37
p14	159	25536.29	13.74	25585.65	1.00	80710.80	3.16	93374.70	3.66	32855.20	1.29
p15	204	1062.44	9.42	1078.29	1.01	5166.97	4.86	2070.83	1.95	1292.43	1.22
p16	129	3526.94	3.54	3526.94	1.00	12970.40	3.68	9201.30	2.61	4865.67	1.38
p17	99	386.23	1.53	386.26	1.00	1043.88	2.70	713.50	1.85	488.35	1.26
p18	99	751.53	1.90	751.53	1.00	1651.83	2.20	1395.56	1.86	1186.79	1.58
p19	201	1344.08	9.04	1351.63	1.01	5397.97	4.02	3063.39	2.28	1824.53	1.36
p20	222	1590.82	13.46	1602.03	1.01	7549.48	4.75	3614.87	2.27	2118.70	1.33

4.2.2 High cost 3-matchings

As we can see in Table 4.3, even with small sets of points ($n \leq 51$), the computation of some maximization instances took several hours (even a few days) on a computer with two AMD Opteron 6174 at 2.2GHz and 128GB RAM; whilst our heuristics took a negligible amount of time (thus not listed in the tables) for the same instances. Again, the performance of our heuristics can be read from the ratio column. In this case, the quotient is at most 1, but again the closer it is to 1, the better. Table 4.3 highlights in bold the best results obtained by our heuristics, always from the Windrose heuristic. It is not clear whether any of our three heuristics is in fact an approximation algorithm.

Table 4.2: Minimum cost and heuristics.

Case	n	minimum	$\text{time}_{[s]}$	Windrose	Ratio	ConvHull	Ratio	Guillotine	Ratio
f21	21	159.73	0.07	212.40	1.330	221.28	1.385	**182.64**	1.143
f27	27	8011.08	0.08	10326.00	1.289	13758.40	1.717	**9578.44**	1.196
f33	33	255.69	0.09	459.28	1.796	350.74	1.372	**299.38**	1.171
39a	39	783.96	0.31	1629.76	2.079	1705.26	2.175	**903.88**	1.153
39b	39	826.14	0.51	1651.56	1.999	1283.06	1.553	**1002.43**	1.213
39c	39	959.38	0.10	1670.49	1.741	1531.89	1.597	**1192.85**	1.243
39d	39	781.22	0.29	1832.70	2.346	1303.39	1.668	**845.48**	1.082
39e	39	872.35	0.70	1427.32	1.636	1471.62	1.687	**948.66**	1.087
f39	39	282.31	0.92	**292.98**	1.038	454.32	1.609	489.56	1.734
42a	42	949.09	0.43	2092.11	2.204	1563.40	1.647	**1022.72**	1.078
42b	42	860.59	0.63	1697.77	1.973	1644.05	1.910	**951.38**	1.105
45a	45	1013.14	0.19	2298.06	2.268	1775.77	1.753	**1319.36**	1.302
45b	45	985.60	0.78	1790.62	1.817	1847.84	1.875	**1153.54**	1.170
48a	48	996.61	0.10	2254.78	2.262	1609.26	1.615	**1220.06**	1.224
48b	48	967.83	0.25	2155.98	2.228	2093.32	2.163	**1381.74**	1.428
51a	51	983.55	0.51	2550.64	2.593	1866.47	1.898	**1131.93**	1.151
51b	51	1003.82	0.60	2140.50	2.132	1927.80	1.920	**1029.04**	1.025
eil51	51	265.61	0.68	532.57	2.005	421.18	1.586	**323.10**	1.216
man	84	1007.11	2.53	2693.77	2.675	2858.21	2.838	**1309.44**	1.300
f99	99	386.23	1.58	1043.88	2.703	713.49	1.847	**488.35**	1.264
120a	120	1508.72	10.52	5125.51	3.397	3336.75	2.212	**1926.33**	1.277
h04	129	3526.94	3.52	12970.40	3.678	9201.30	2.609	**4865.67**	1.380
201a	201	1945.09	14.23	9164.72	4.712	4327.80	2.225	**2424.20**	1.246
240a	240	2068.67	12.99	11089.60	5.361	5760.70	2.785	**2871.38**	1.388
300a	300	2202.32	56.24	13835.30	6.282	5790.96	2.629	**3082.12**	1.399
360a	360	2520.41	281.49	17297.90	6.863	7458.42	2.959	**3700.60**	1.468
h07	441	30979.16	116.22	127301.00	4.109	104987.00	3.389	**49480.10**	1.597
510a	510	2999.25	341.95	23933.90	7.980	9170.59	3.058	**4588.09**	1.530
h08	573	4106.92	153.54	23507.60	5.724	9835.20	2.395	**6314.24**	1.537
rat783	783	5269.62	558.24	36178.10	6.865	15074.40	2.861	**8320.26**	1.579
h10	1002	148206.63	8746.71	1133310.00	7.647	553990.00	3.738	**210241.00**	1.419

Table 4.3: Maximum cost and heuristics.

Case	n	maximum	time$_{[hr]}$	Windrose	Ratio	ConvHull	Ratio	Guillotine	Ratio
f21	21	492.19	0.01	**418.00**	0.849	307.54	0.625	277.77	0.564
f27	27	39330.22	0.03	**35834.30**	0.911	24311.40	0.618	18078.50	0.460
f33	33	1136.34	0.32	**887.77**	0.781	712.82	0.627	601.63	0.529
39a	39	4997.58	2.41	**4107.73**	0.822	2774.23	0.555	2172.61	0.435
39b	39	4548.58	2.31	**3077.47**	0.677	2496.36	0.549	1632.18	0.359
39c	39	4410.10	1.75	**3726.14**	0.845	2725.35	0.618	1887.40	0.428
39d	39	4728.09	3.06	**3728.20**	0.789	2974.18	0.629	2330.61	0.493
39e	39	4804.63	2.07	**4205.22**	0.875	3068.20	0.639	1783.95	0.371
f39	39	3804.50	0.54	**3320.16**	0.873	1998.54	0.525	994.54	0.261
42a	42	5026.50	8.71	**3818.60**	0.760	3225.15	0.642	2588.07	0.515
42b	42	4918.73	8.69	**3524.74**	0.717	3433.42	0.698	2502.66	0.509
45a	45	5710.43	12.29	**5093.17**	0.892	3319.70	0.581	2416.68	0.423
45b	45	6009.25	8.68	**4676.96**	0.778	3772.16	0.628	2255.34	0.375
48a	48	5861.28	57.81	**5097.48**	0.870	3665.41	0.625	2608.46	0.445
48b	48	5932.78	38.58	**4947.42**	0.834	3583.77	0.604	2484.23	0.419
51a	51	6266.31	52.05	**5150.67**	0.822	4042.01	0.645	2004.47	0.320
51b	51	6276.55	48.88	**4489.25**	0.715	3770.45	0.601	2497.22	0.398
eil51	51	1247.13	89.29	**1098.85**	0.881	749.35	0.601	545.16	0.437

Chapter 5

Conclusions and Future Work

In this work we have approached both the minimization and maximization versions of the non-crossing Euclidean 3-matching problem. Both of them are believed to be NP-hard problems because their combinatorial counterparts are known to be NP-complete even for metric costs. However, the non-crossing maximization version seems to be harder than the minimization version. This appears to be true for many geometric network optimization problems.

Our first approach was based on proposing integer programming formulations for the minimization version. There were two previously known formulations and we showed that the linear programming relaxations of the two previously known formulations are equivalent, i.e., they give the same minimum value. We also proposed three new formulations. The main difference is that our formulations use one variable per edge, while the previously known formulations used two variables per edge. We also studied their linear programming relaxations. We conjecture that at least two of our relaxations (Pair and Quad) give a tighter lower bound. We also conjecture that the previously known relaxations and our Triple linear programming relaxation give a planar graph. In order to obtain models for the maximization version, we had to add constraints to avoid crossings.

Our second approach was based on proposing heuristics for our two problems. The main reason for this was that solving our models to optimality is very time consuming, particularly so for the maximization version. We have proposed three geometric heuristics specific to these problems, each of them running in fractions of a second. Although our heuristics do not give the best possible results, the time needed to run previously known heuristics exceeds the time to obtain the optimal values using state-of-the-art solvers. Furthermore, our heuristics were easily adapted for both minimization and maximization versions.

Our future work on these problems shall include proving the conjectures mentioned above, as well as designing approximation algorithms for our problems.

Appendix A

Source code

A.1 Johnsson, Magyar, and Nevalainen (1998) model generator

```php
<?php
  // Modelo 1998

  $puntos = [ ];
  $x = null;
  $y = null;

  while (fscanf(STDIN, "%f%f", $x, $y) == 2) {
    $puntos[] = [ $x, $y ];
  }

  $n = count($puntos);

  function distancia($v1, $v2)
  {
    $t1 = $v1[0] - $v2[0];
    $t2 = $v1[1] - $v2[1];

    return sqrt($t1 * $t1 + $t2 * $t2);
  }

  echo "minimize\n";
  $pesos = [ ];

```

```
25  for ($i = 0; $i < $n; ++$i) {
26    for ($j = $i + 1; $j < $n; ++$j) {
27      $coeficiente = distancia($puntos[$i], $puntos[$j]);
28      $pesos[] = "{$coeficiente} arista_{$i}_{$j}";
29      $pesos[] = "{$coeficiente} arista_{$j}_{$i}";
30    }
31  }
32
33  echo "\t", implode(' + ', $pesos), "\n";
34
35  // restricciones
36
37  echo "subject to\n";
38
39  for ($i = 0; $i < $n; ++$i) {
40    $aristas_entran = [ ];
41    $aristas_salen = [ ];
42    for ($j = 0; $j < $n; ++$j) {
43      if ($i == $j) {
44        continue;
45      }
46
47      $aristas_entran[] = "arista_{$j}_{$i}";
48      $aristas_salen[] = "arista_{$i}_{$j}";
49    }
50
51    $suma_entran = implode(' - ', $aristas_entran);
52    $suma_salen = implode(' - ', $aristas_salen);
53
54
55    //aristas de que salen y entran al vetices i
56
57    echo "\tvertice_{$i}_entran - {$suma_entran} = 0 \n";
58    echo "\tvertice_{$i}_salen - {$suma_salen} = 0 \n";
59  }
60
61  // restriccion:
62    for ($i = 0; $i < $n; ++$i){
63      echo "\t0.5 vertice_{$i}_entran + vertice_{$i}_salen = 1 \n";
64    }
65
66
67  // variables
```

```
68
69    echo "integers\n";
70
71    for ($i = 0; $i < $n; ++$i) {
72      echo "\tvertice_{$i}_entran \n";
73      echo "\tvertice_{$i}_salen \n";
74    }
75
76
77    echo "binary\n";
78    for ($i = 0; $i < $n; ++$i) {
79      for ($j = $i + 1; $j < $n; ++$j) {
80        echo "\tarista_{$i}_{$j}\n";
81        echo "\tarista_{$j}_{$i}\n";
82      }
83    }
84    echo "end\n";
85
86  ?>
```

codigos/modelo1998.php

A.2 Johnsson, Magyar, and Nevalainen (1999) model generator

```php
<?php
  // Modelo 1999

  $puntos = [ ];
  $x = null;
  $y = null;

  while (fscanf(STDIN, "%f%f", $x, $y) == 2) {
    $puntos[] = [ $x, $y ];
  }

  $n = count($puntos);

  function distancia($v1, $v2)
  {
    $t1 = $v1[0] - $v2[0];
    $t2 = $v1[1] - $v2[1];

    return sqrt($t1 * $t1 + $t2 * $t2);
  }

  echo "minimize\n";
  $pesos = [ ];

  for ($i = 0; $i < $n; ++$i) {

    for ($j = $i + 1; $j < $n; ++$j) {
      $coeficiente = distancia($puntos[$i], $puntos[$j]);
      $pesos[] = "{$coeficiente} arista_{$i}_{$j}";
      $pesos[] = "{$coeficiente} arista_{$j}_{$i}";
    }
  }

  echo "\t", implode(' + ', $pesos), "\n";

  // restricciones

  echo "subject to\n";
```

```
39
40  // restriccion: x_ij = 1
41  for ($i = 0; $i < $n; ++$i) {
42    $aristas_vertice = [ ];
43    for ($j = 0; $j < $n; ++$j) {
44      $aristas_vertice[] = "arista_{$i}_{$j}";
45    }
46
47    $suma_entran = implode(' + ', $aristas_vertice);
48
49    //aristas de que salen y entran al vetices i
50    echo "\t$suma_entran = 1 \n";
51  }
52
53  // restriccion : x_ii = n/3 vertices centrales
54  $aristas_vertice = [ ];
55  for ($i = 0; $i < $n; ++$i){
56    $aristas_vertice[]= "arista_{$i}_{$i}";
57  }
58  $suma_vertices = implode(' + ', $aristas_vertice);
59  echo "\t$suma_vertices - n/3 = 0\n";
60
61  // restriccion : x_rs = 2 x_rr
62
63  for ($j = 0; $j < $n; ++$j){
64    $aristas_vertice = [ ];
65    for ($i = 0; $i < $n; ++$i){
66      if ($i == $j){
67        continue;
68      }
69      $aristas_vertice[] = "arista_{$i}_{$j}";
70    }
71    $suma_vertices = implode(' + ', $aristas_vertice);
72    echo "\t$suma_vertices - 2 arista_{$j}_{$j} = 0\n";
73  }
74
75  // variables
76
77  echo "binary\n";
78  for ($i = 0; $i < $n; ++$i) {
79    echo "\tarista_{$i}_{$i}\n";
80    for ($j = $i + 1; $j < $n; ++$j) {
81      echo "\tarista_{$i}_{$j}\n";
```

```
82        echo "\tarista_{$j}_{$i}\n";
83      }
84    }
85
86    echo "end\n";
87
88  ?>
```

codigos/modelo1999.php

A.3 Triplet integer programming model generator

```php
<?php
  $puntos = [ ];
  $x = null;
  $y = null;

  while (fscanf(STDIN, "%f%f", $x, $y) == 2) {
    $puntos[] = [ $x, $y ];
  }

  $n = count($puntos);

  function distancia($v1, $v2)
  {
    $t1 = $v1[0] - $v2[0];
    $t2 = $v1[1] - $v2[1];

    return sqrt($t1 * $t1 + $t2 * $t2);
  }

  echo "minimize\n";
  $pesos = [ ];

  for ($i = 0; $i < $n; ++$i) {
    for ($j = $i + 1; $j < $n; ++$j) {
      $coeficiente = distancia($puntos[$i], $puntos[$j]);
      $pesos[] = "{$coeficiente} arista_{$i}_{$j}";
    }
  }

  echo "\t", implode(' + ', $pesos), "\n";

  // restricciones

  echo "subject to\n";

  // restricciones: suma(v) >= 1
  $suma_vertice = [ ];
  for ($i = 0; $i < $n; ++$i) {
    $actuales = [ ];

```

```
41    for ($j = 0; $j < $n; ++$j) {
42      if ($i == $j) {
43        continue;
44      }
45
46      $actuales[] = ($i < $j ? "arista_{$i}_{$j}" : "arista_{$j}_{$i}");
47    }
48
49    $suma = implode(' - ', $actuales);
50    $suma_vertice[] = $suma;
51
52    echo "\tvertice_{$i} - {$suma} = 0 \n";
53  }
54
55  /** restricciones:  suma(r,s,t) <= 3 (2 - ~suma(r,s,t))
56            suma(r,s,t) >= 3/2 (2 - ~suma(r,s,t))
57  */
58
59  for ($i = 0; $i < $n; ++$i){
60    $vertices_conectados = [ ];
61    for ($j = $i + 1; $j < $n; ++$j){
62      $vertices_conectados[] = ($i < $j ? "3 arista_{$i}_{$j}" : "3
   arista_{$j}_{$i}");
63      for($k = $j + 1; $k < $n; ++$k){
64        $vertices_conectados[] = ($i < $k ? "3 arista_{$i}_{$k}" : "3
   arista_{$k}_{$i}");
65        $vertices_conectados[] = ($j < $k ? "3 arista_{$j}_{$k}" : "3
   arista_{$k}_{$j}");
66        $suma = implode(' + ', $vertices_conectados);
67        $arista1 = ($i < $j ? "arista_{$i}_{$j}" : "arista_{$j}_{$i}");
68        $arista2 = ($i < $k ? "arista_{$i}_{$k}" : "arista_{$k}_{$i}");
69        $arista3 = ($j < $k ? "arista_{$j}_{$k}" : "arista_{$k}_{$j}");
70        echo "\tvertice_{$i} + vertice_{$j}  + vertice_{$k} - 2 $arista1
   - 2 $arista2 - 2 $arista3 + $suma <= 6 \n";
71        echo "\t2 vertice_{$i} + 2 vertice_{$j}  + 2 vertice_{$k} - 4
   $arista1  - 4 $arista2 - 4 $arista3 + $suma >= 6 \n";
72
73        for($quita = 0; $quita < 2; ++$quita){
74          array_pop($vertices_conectados);
75        }
76      }
77      array_pop($vertices_conectados);
78    }
```

```
79   }
80
81   // variables
82
83   echo "bounds\n";
84
85   for ($i = 0; $i < $n; ++$i) {
86     echo "\tvertice_{$i} >= 1 \n";
87   }
88
89   echo "integers\n";
90
91   for ($i = 0; $i < $n; ++$i) {
92     echo "\tvertice_{$i} \n";
93   }
94
95   echo"binary\n";
96
97   for ($i = 0; $i < $n; ++$i) {
98     for ($j = $i + 1; $j < $n; ++$j) {
99       echo "\tarista_{$i}_{$j}\n";
100    }
101  }
102
103  echo "end\n";
104 ?>
```

codigos/tripletas.php

A.4 Pair integer programming model generator

```php
<?php
  $puntos = [ ];
  $x = null;
  $y = null;

  while (fscanf(STDIN, "%f%f", $x, $y) == 2) {
    $puntos[] = [ $x, $y ];
  }

  $n = count($puntos);

  function distancia($v1, $v2)
  {
    $t1 = $v1[0] - $v2[0];
    $t2 = $v1[1] - $v2[1];

    return sqrt($t1 * $t1 + $t2 * $t2);
  }

  echo "minimize\n";
  $pesos = [ ];

  for ($i = 0; $i < $n; ++$i) {
    for ($j = $i + 1; $j < $n; ++$j) {
      $coeficiente = distancia($puntos[$i], $puntos[$j]);
      $pesos[] = "{$coeficiente} arista_{$i}_{$j}";
    }
  }

  echo "\t", implode(' + ', $pesos), "\n";

  // restricciones

  echo "subject to\n";

  // restricciones: suma(v) >= 1
  $suma_vertice = [ ];
  for ($i = 0; $i < $n; ++$i) {
    $actuales = [ ];

```

```
41      for ($j = 0; $j < $n; ++$j) {
42        if ($i == $j) {
43          continue;
44        }
45
46        $actuales[] = ($i < $j ? "arista_{$i}_{$j}" : "arista_{$j}_{$i}");
47      }
48
49      $suma = implode(' - ', $actuales);
50      $suma_vertice[] = $suma;
51
52      echo "\tvertice_{$i} - {$suma} = 0 \n";
53      //echo "\tvertice_{$i} >= 1\n";
54    }
55
56    // restricciones: suma(x) + suma(y) >= 2 + xy
57    //           suma(x) + suma(y) <= 4 - xy
58
59    for ($i = 0; $i < $n; ++$i) {
60      for ($j = $i + 1; $j < $n; ++$j) {
61        $actual = ($i < $j ? "arista_{$i}_{$j}" : "arista_{$j}_{$i}");
62
63        echo "\tvertice_{$i} + vertice_{$j} - {$actual} >= 2 \n";
64        echo "\tvertice_{$i} + vertice_{$j} + {$actual} <= 4 \n";
65      }
66    }
67
68    // variables
69
70    echo "bounds\n";
71
72    for ($i = 0; $i < $n; ++$i) {
73      echo "\tvertice_{$i} >= 1 \n";
74    }
75
76    echo "integers\n";
77
78    for ($i = 0; $i < $n; ++$i) {
79      echo "\tvertice_{$i} \n";
80    }
81
82
83    echo "binary\n";
```

```
84
85    for ($i = 0; $i < $n; ++$i) {
86      for ($j = $i + 1; $j < $n; ++$j) {
87        echo "\tarista_{$i}_{$j}\n";
88      }
89    }
90
91    echo "end\n";
92  ?>
```

codigos/parejas.php

A.5 Quad integer programming model generator

```php
<?php
  $puntos = [ ];
  $x = null;
  $y = null;

  while (fscanf(STDIN, "%f%f", $x, $y) == 2) {
    $puntos[] = [ $x, $y ];
  }

  $n = count($puntos);

  function distancia($v1, $v2)
  {
    $t1 = $v1[0] - $v2[0];
    $t2 = $v1[1] - $v2[1];

    return sqrt($t1 * $t1 + $t2 * $t2);
  }

  echo "minimize\n";
  $pesos = [ ];

  for ($i = 0; $i < $n; ++$i) {
    for ($j = $i + 1; $j < $n; ++$j) {
      $coeficiente = distancia($puntos[$i], $puntos[$j]);
      $pesos[] = "{$coeficiente} arista_{$i}_{$j}";
    }
  }

  echo "\t", implode(' + ', $pesos), "\n";

  // restricciones

  echo "subject to\n";

  // restricciones: suma(v) >= 1
  $suma_vertice = [ ];
  for ($i = 0; $i < $n; ++$i) {
    $actuales = [ ];

```

```
41      for ($j = 0; $j < $n; ++$j) {
42        if ($i == $j) {
43          continue;
44        }
45
46        $actuales[] = ($i < $j ? "arista_{$i}_{$j}" : "arista_{$j}_{$i}");
47      }
48
49      $suma = implode(' - ', $actuales);
50      $suma_vertice[] = $suma;
51
52      echo "\tvertice_{$i} - {$suma} = 0 \n";
53    }
54
55    // restricciones: suma(x,y) >= 1
56
57    for ($i = 0; $i < $n; ++$i){
58      for ($j = $i + 1; $j < $n; ++$j){
59        $arista = ($i < $j ? "arista_{$i}_{$j}" : "arista_{$j}_{$i}");
60        echo "\tvertice_{$i} + vertice_{$j} - 2 $arista >= 1 \n";
61      }
62    }
63
64    // restricciones: ~suma(x,y,r,s) >= 2
65
66    for ($i = 0; $i < $n; ++$i){
67      for ($j = $i + 1; $j < $n; ++$j){
68        for ($k = $j + 1; $k < $n; ++$k){
69            for ($l = $k + 1; $l < $n; ++$l){
70              echo "\tarista_{$i}_{$j} + arista_{$i}_{$k} + arista_{$i}_{$l}
      + arista_{$j}_{$k} + arista_{$j}_{$l} + arista_{$k}_{$l} <= 2\n";
71            }
72        }
73      }
74    }
75    // variables
76
77    echo "bounds\n";
78
79    for ($i = 0; $i < $n; ++$i) {
80      echo "\tvertice_{$i} >= 1 \n";
81    }
82
```

```
83   echo "integers\n";
84
85   for ($i = 0; $i < $n; ++$i) {
86     echo "\tvertice_{$i} \n";
87   }
88
89   echo "binary\n";
90
91   for ($i = 0; $i < $n; ++$i) {
92     for ($j = $i + 1; $j < $n; ++$j) {
93       echo "\tarista_{$i}_{$j}\n";
94     }
95   }
96
97   echo "end\n";
98 ?>
```

codigos/cuadrupletas.php

A.6 Windrose heuristic

```
#include <algorithm>
#include <cmath>
#include <iostream>
#include <utility>
#include <vector>

double distancia(std::pair<double, double>& P1, const std::pair<double, double>& P2){
    double t1 = P2.first - P1.first;
    double t2 = P2.second - P1.second;

    return std::sqrt(t1 * t1 + t2 * t2);
}

bool ordenaY (const std::pair<float, float> P1, const std::pair<float, float> P2){
    if (P1.second == P2.second){
        return P1.first < P2.first;
    }

    return P1.second < P2.second;
}

bool ordenaX (const std::pair<float, float> P1, const std::pair<float, float> P2){
    if (P1.first == P2.first){
        return P1.second < P2.second;
    }

    return P1.first < P2.first;
}

bool ordenaD1 (const std::pair<float, float> P1, const std::pair<float, float> P2){
    if ((P1.first + P1.second) == (P2.first + P2.second)){
        return (P1.first - P1.second) < (P2.first - P2.second);
    }

    return (P1.first + P1.second) < (P2.first + P2.second);
}
```

```
37
38 bool ordenaD2 (const std::pair<float, float> P1, const std::pair<float,
       float> P2){
39     if ((P1.first - P1.second) == (P2.first - P2.second)){
40         return (P1.first + P1.second) < (P2.first + P2.second);
41     }
42
43     return (P1.first - P1.second) < (P2.first - P2.second);
44 }
45
46 bool criterio (const std::pair<double, std::pair<int, int>>  P1, std::pair
       <double, std::pair<int, int>> P2){
47     return P1.first > P2.first;
48 }
49
50 int main( )
51 {
52     std::pair<double, double> punto;
53     std::vector<std::pair<double, double>> Puntos;
54
55     while(std::cin >> punto.first >> punto.second){
56         Puntos.push_back(punto);
57     }
58     std::sort(Puntos.begin( ), Puntos.end( ), ordenaY);
59
60     double costoTotal = 0;
61     std::vector<std::pair<int, int>> aristas;
62
63
64     for (int i = 0; i < Puntos.size( ); i += 3){
65         std::vector<std::pair<double, std::pair<int, int>>> trespuntos;
66         std::pair<double, std::pair<int, int>> tmp;
67         tmp.first = distancia(Puntos[i], Puntos[i + 1]);
68         tmp.second.first = i;
69         tmp.second.second = i + 1;
70         trespuntos.push_back(tmp);
71
72         tmp.first = distancia(Puntos[i], Puntos[i + 2]);
73         tmp.second.first = i;
74         tmp.second.second = i + 2;
75         trespuntos.push_back(tmp);
76
77         tmp.first = distancia(Puntos[i + 1], Puntos[i + 2]);
```

```
78          tmp.second.first = i + 1;
79          tmp.second.second = i + 2;
80          trespuntos.push_back(tmp);
81
82          std::sort(trespuntos.begin( ), trespuntos.end( ), criterio);
83          aristas.push_back(trespuntos[0].second);
84          aristas.push_back(trespuntos[1].second);
85          costoTotal += trespuntos[0].first + trespuntos[1].first;
86
87      }
88      std::cout << costoTotal << '\n';
89      std::cout << Puntos.size( ) << '\n';
90      for (auto& i : Puntos){
91          std::cout << i.first << ' ' << i.second << '\n';
92      }
93      for (auto& i : aristas){
94          std::cout << i.first << ' ' << i.second << '\n';
95      }
96
97      return 0;
98  }
```

codigos/orientacion.cpp

A.7 ConvHull heuristic

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <utility>

template<typename T>
inline T producto_cruz(const std::pair<T, T>& a, const std::pair<T, T>& b,
     const std::pair<T, T>& c)
{
    return (b.first - a.first) * (c.second - a.second) - (b.second - a.
    second) * (c.first - a.first);
}

template<typename RI1, typename RI2>
inline RI2 cerco_parcial(RI1 ai, RI1 af, RI2 bi)
{
    auto bw = bi;
    for (; ai != af; *bw++ = *ai++) {
       while (bw - bi >= 2 && producto_cruz(*(bw - 2), *(bw - 1), *ai) <=
    0) {
          --bw;
       }
    }

    return bw;
}

template<typename RI, typename OI>
inline OI cerco_convexo(RI ai, RI af, OI wi)
{
    typename std::iterator_traits<RI>::value_type temp[af - ai];
    wi = std::copy(temp, cerco_parcial(ai, af, temp) - 1, wi);
    wi = std::copy(temp, cerco_parcial(std::make_reverse_iterator(af), std
    ::make_reverse_iterator(ai), temp) - 1, wi);

    return wi;
}


inline void imprime(const std::vector<std::pair<int, int>>& v, int dim)
```

```
37 {
38    char tablero[dim][dim];
39    std::fill(&tablero[0][0], &tablero[dim][0], '.');
40 
41    for (auto par : v) {
42       tablero[par.first][par.second] = '#';
43    }
44 
45    for (int i = 0; i < dim; ++i) {
46       for (int j = 0; j < dim; ++j) {
47          std::cout << tablero[i][j];
48       }
49 
50       std::cout << '\n';
51    }
52 }
53 
54 double distancia(std::pair<double, double>& P1, const std::pair<double,
       double>& P2){
55     double t1 = P2.first - P1.first;
56     double t2 = P2.second - P1.second;
57 
58     return std::sqrt(t1 * t1 + t2 * t2);
59 }
60 
61 
62 bool ordena (const std::pair<int, double>& valor1, const std::pair<int,
       double>& valor2){
63    return valor1.second > valor2.second;
64 }
65 
66 int main( )
67 {
68    /*constexpr auto D = 70;
69 
70    std::vector<std::pair<int, int>> v1;
71    for (int i = 0; i < D; ++i) {
72       for (int j = 0; j < D; ++j) {
73          if (std::rand( ) % 8 == 0) {
74             v1.emplace_back(i, j);
75          }
76       }
77    }*/
```

```
78    std::vector<std::pair<double, double>> Resultado;
79
80    std::vector<std::pair<double, double>> v1;
81    double x, y;
82    double costo = 0;
83    while (std::cin >> x >> y){
84
85       v1.emplace_back(x, y);
86    }
87
88    while (!v1.empty( )){
89       std::sort(v1.begin( ), v1.end( ));
90       std::vector<std::pair<double, double>> v2;
91       cerco_convexo(v1.begin( ), v1.end( ), std::back_inserter(v2));
92
93       std::vector<double> aristas;
94       aristas.resize(v2.size( ));
95
96       for (int i = 0; i < v2.size( ) - 1; ++i){
97          aristas[i] =  distancia(v2[i], v2[i + 1]);
98       }
99
100      aristas[ aristas.size( ) - 1 ] =  distancia(v2[ v2.size( ) - 1 ], v2
    [0]);
101
102      std::vector<std::pair<int, double>> acoplamientoMax;
103      acoplamientoMax.resize(v2.size( ));
104
105      for (int i = 0; i < acoplamientoMax.size( ) - 1; ++i){
106         acoplamientoMax[i].first = i;
107         acoplamientoMax[i].second = aristas[i] + aristas[i + 1];
108      }
109      acoplamientoMax[ acoplamientoMax.size( ) - 1 ].first =
    acoplamientoMax.size( ) - 1;
110      acoplamientoMax[ acoplamientoMax.size( ) - 1 ].second = aristas[
    acoplamientoMax.size( ) - 1 ] + aristas[0];
111
112      std::sort(acoplamientoMax.begin( ), acoplamientoMax.end( ), ordena);
113
114      costo+= acoplamientoMax[0].second;
115      int empieza = acoplamientoMax[0].first;
116
117      for (int i = 0; i < 3; ++i){
```

```
118             for (int j = 0; j < v1.size( ); ++j){
119                 if (v2[empieza] == v1[j]){
120                     std::swap(v1[j], v1[v1.size( ) -1]);
121                     v1.pop_back( );
122                     break;
123                 }
124             }
125 
126             Resultado.push_back(v2[empieza++]);
127             empieza %= v2.size( );
128         }
129     }
130     std::cout << costo << '\n';
131     std::cout << Resultado.size( ) << '\n';
132 
133     for (auto& i : Resultado){
134         std::cout << i.first << '\t' << i.second << '\n';
135     }
136 }
```

codigos/cerco_convexo.cpp

A.8 Guillotine heuristic

```
#include <algorithm>
#include <cmath>
#include <iostream>
#include <utility>
#include <vector>

std::vector<std::pair<int, int>> Solucion;
double costo = 0;

double distancia(const std::pair<double, double>& P1, const std::pair<
    double, double>& P2){
     double t1 = P2.first - P1.first;
     double t2 = P2.second - P1.second;

     return std::sqrt(t1 * t1 + t2 * t2);
}
bool ordena (const std::pair<int, double>& valor1, const std::pair<int,
    double>& valor2){
    return valor1.second < valor2.second;
}

bool ordenaX (const std::pair<double,double> P1, const std::pair<double,
    double> P2){
     if (P1.first == P2.first){
         return P1.second < P2.second;
     }

     return P1.first < P2.first;
}

bool ordenaY (const std::pair<double,double> P1, const std::pair<double,
    double> P2){
     if (P1.second == P2.second){
         return P1.first < P2.first;
     }

     return P1.second < P2.second;
}

```

```
36 void evalua(const std::pair<double,double>& x, const std::pair<double,
       double>& y, const std::pair<double,double>& z){
37     std::vector<std::pair<int,double>> distancias;
38     distancias.emplace_back(0, distancia(x , y));
39     distancias.emplace_back(1, distancia(y , z));
40     distancias.emplace_back(2, distancia(z , x));
41 
42     std::sort(distancias.begin( ), distancias.end( ), ordena);
43     for (int i = 0; i < 2; ++i){
44 
45         costo += distancias[i].second;
46         if (distancias[i].first == 0){
47             Solucion.push_back(x);
48         }
49         else if (distancias[i].first == 1){
50             Solucion.push_back(y);
51         }
52         else{
53             Solucion.push_back(z);
54         }
55     }
56 }
57 
58 void L0 (std::vector<std::pair<double,double>> P , int sentido){
59     if ( P.size( ) == 3 ){
60         evalua(P[0], P[1], P[2]);
61         return;
62     }
63 
64     int tam = P.size( ) / 3 - 1;
65     double GAP[tam];
66 
67     if (sentido == 0){
68         std::sort(P.begin( ), P.end( ), ordenaX);
69         for (int i = 0; i < tam; ++i){
70             GAP[i] = P[i * 3 + 3].first - P[i * 3 + 2].first;
71         }
72     }
73     else{
74         std::sort(P.begin( ), P.end( ), ordenaY);
75         for (int i = 0; i < tam; ++i){
76             GAP[i] = P[i * 3 + 3].second - P[i * 3 + 2].second;
77         }
```

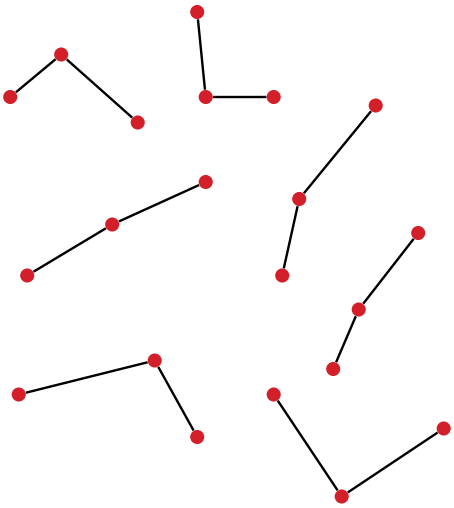
```
78      }
79
80      int corte = std::min_element(GAP, GAP + tam) - GAP;
81
82      std::vector<std::pair<double,double>> hoja1, hoja2;
83
84      hoja1.insert(hoja1.begin( ), P.begin( ), P.begin( ) + (corte * 3 + 3))
        ;
85      hoja2.insert(hoja2.begin( ), P.begin( ) + (corte * 3 + 3), P.end( ));
86
87      if (sentido == 0){
88          L0(hoja1, 1);
89          L0(hoja2, 1);
90      }
91      else{
92          L0(hoja1, 0);
93          L0(hoja2, 0);
94      }
95  }
96
97  int main ( )
98  {
99
100     std::vector<std::pair<double,double>> Puntos;
101     double x, y;
102
103     while(std::cin >> x >> y){
104       Puntos.emplace_back(x, y);
105     }
106     std::cout << Puntos.size( ) << '\n';
107     L0(Puntos, 0);
108     std::cout << costo << '\n';
109 }
```

codigos/guillotina.cpp

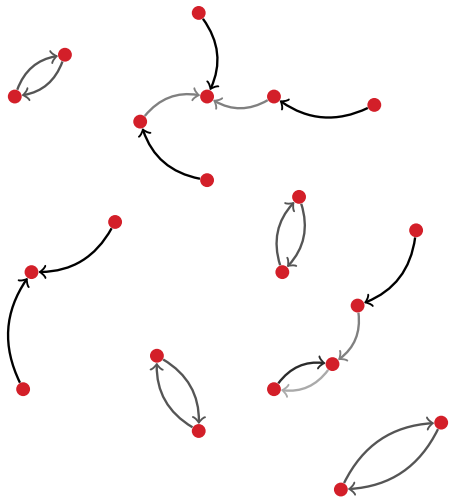
Appendix B

Minimization Instances

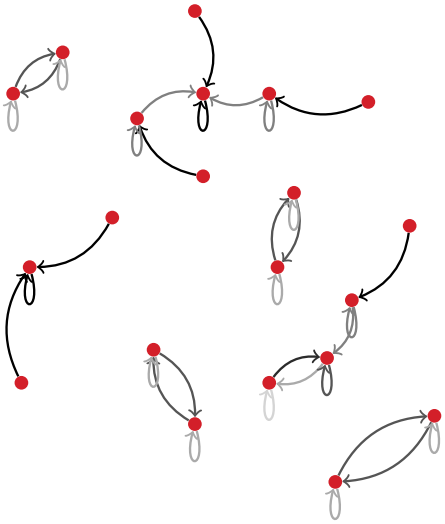
In this appendix, we draw the benchmark instances used for the minization version of the 3-matching problem without crossings. We list the (a) optimal solution, (b) relaxation of the first integer programming model by Johnsson, Magyar, and Nevalainen, (c) relaxation of the second integer programming model by Johnsson, Magyar, and Nevalainen, (d) relaxation of the pair integer programming model, (e) relaxation of the triplet integer programming model and (f) relaxation of the quad integer programming model. The bolder the edge $\overline{rs}$ appears in the drawing, the closer to 1 is the corresponding variable $0 \leq x_{rs} \leq 1$ in the relaxation.



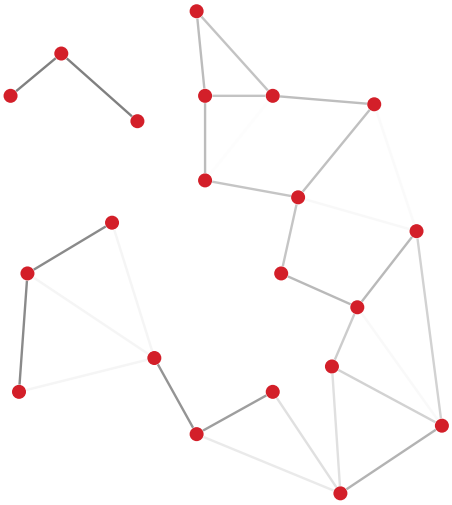
(a)



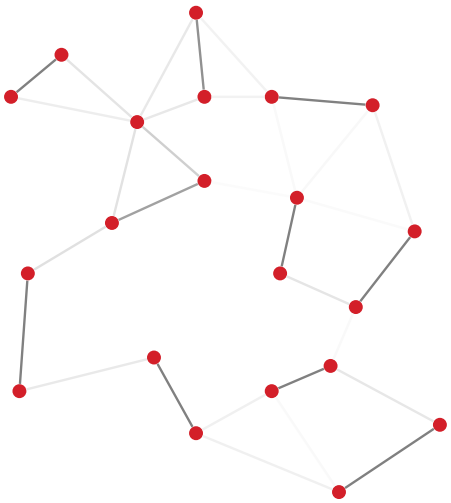
(b)



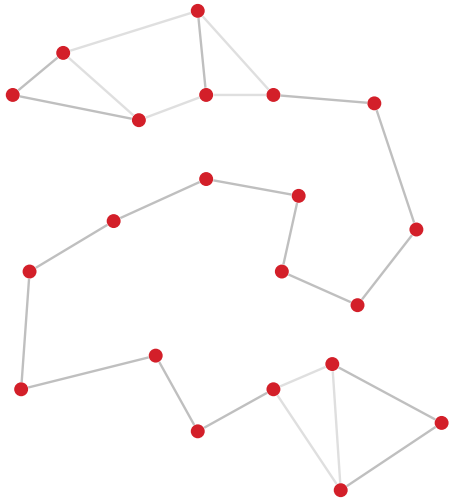
(c)



(d)



(e)



(f)

Figure B.1: Instance f21

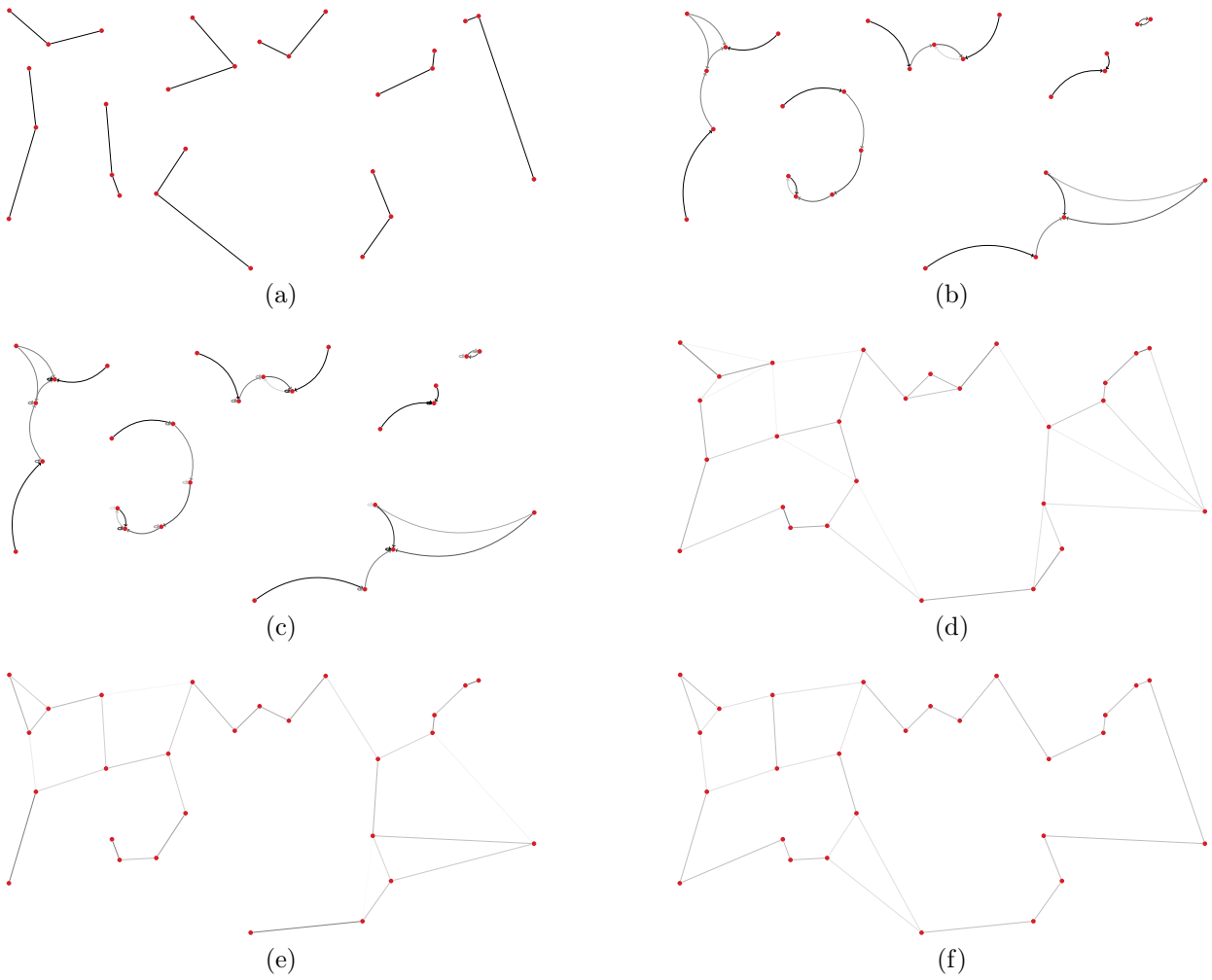
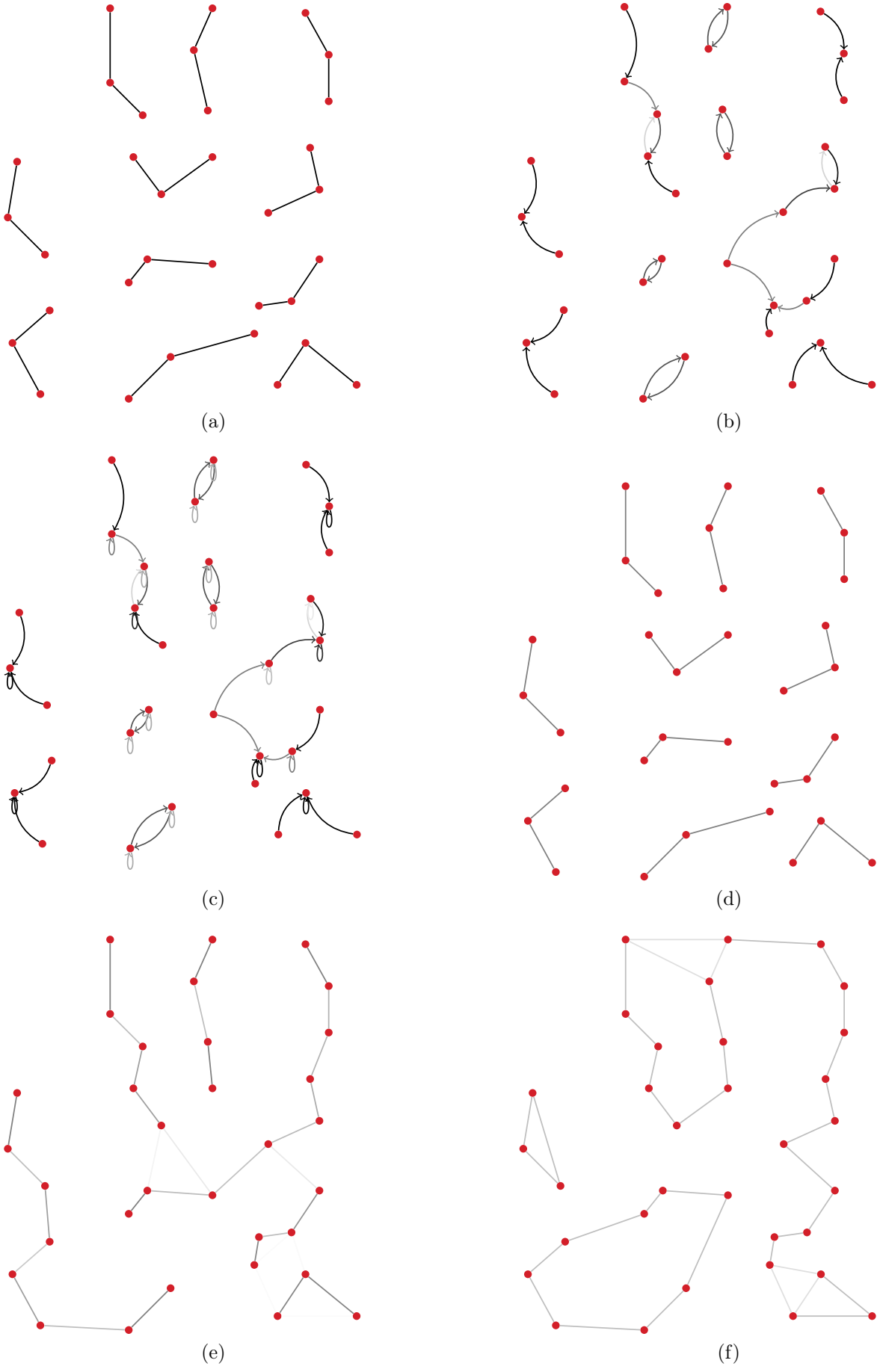
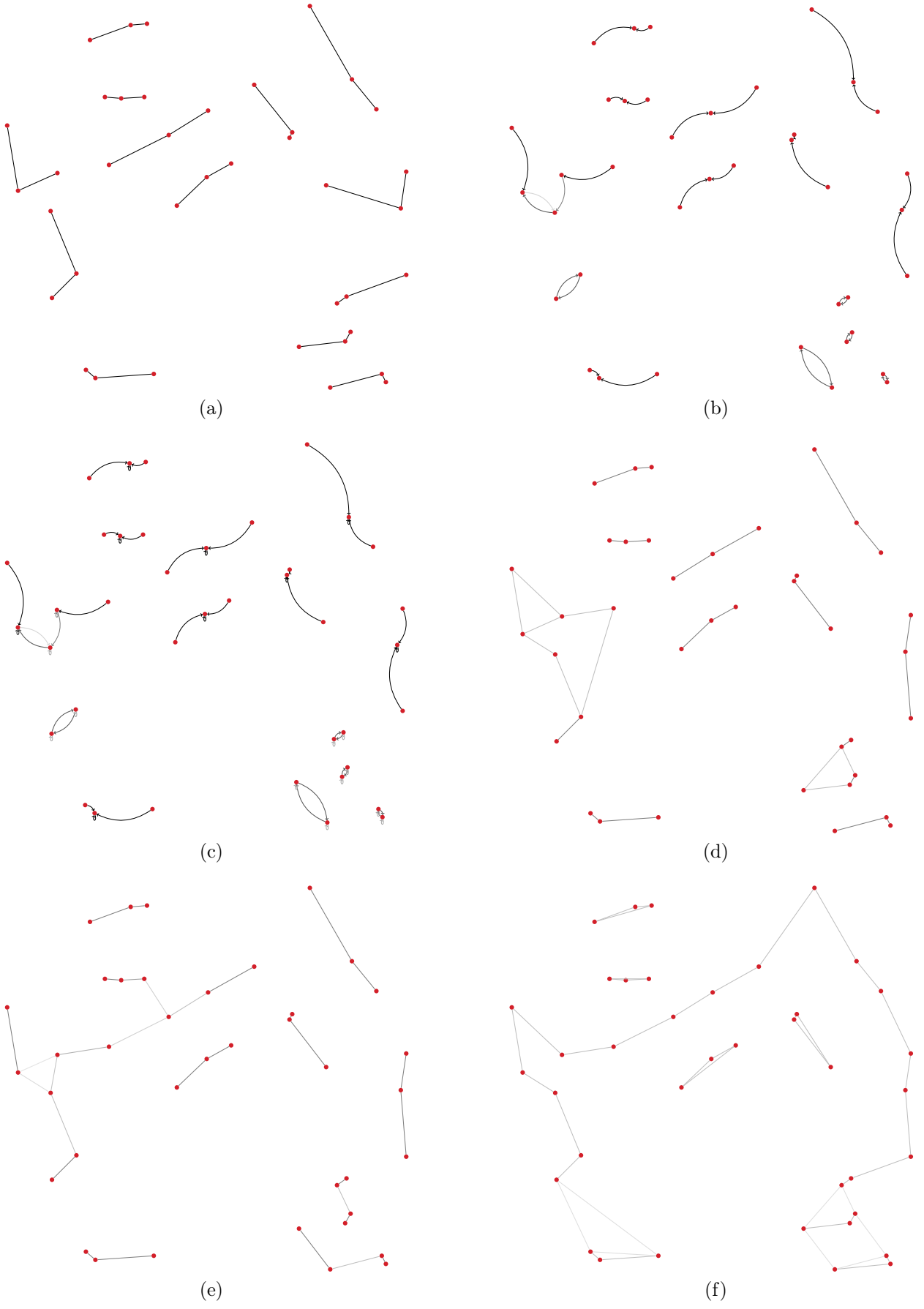


Figure B.2: Instance f27



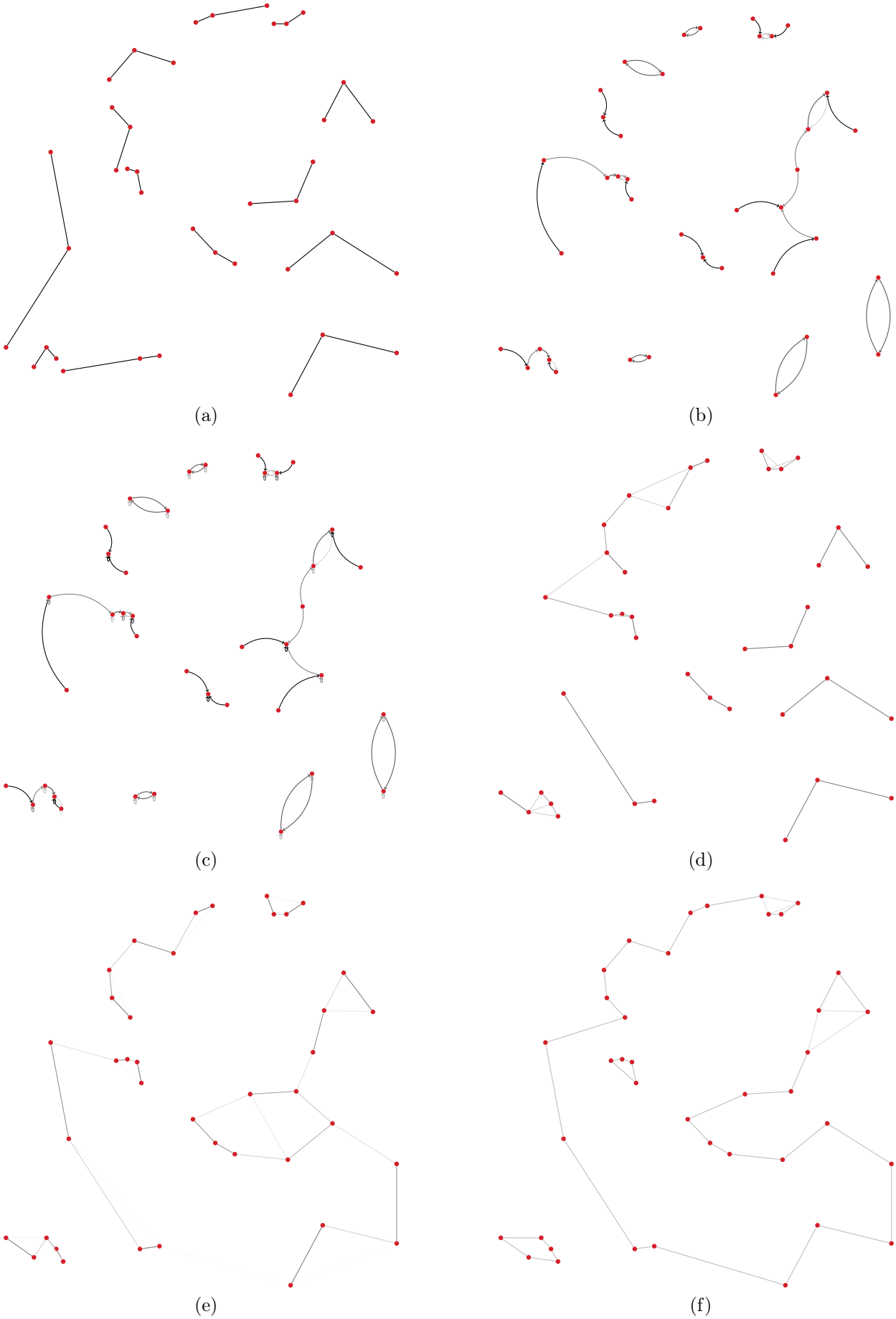
(a) (b) (c) (d) (e) (f)

Figure B.3: Instance f33



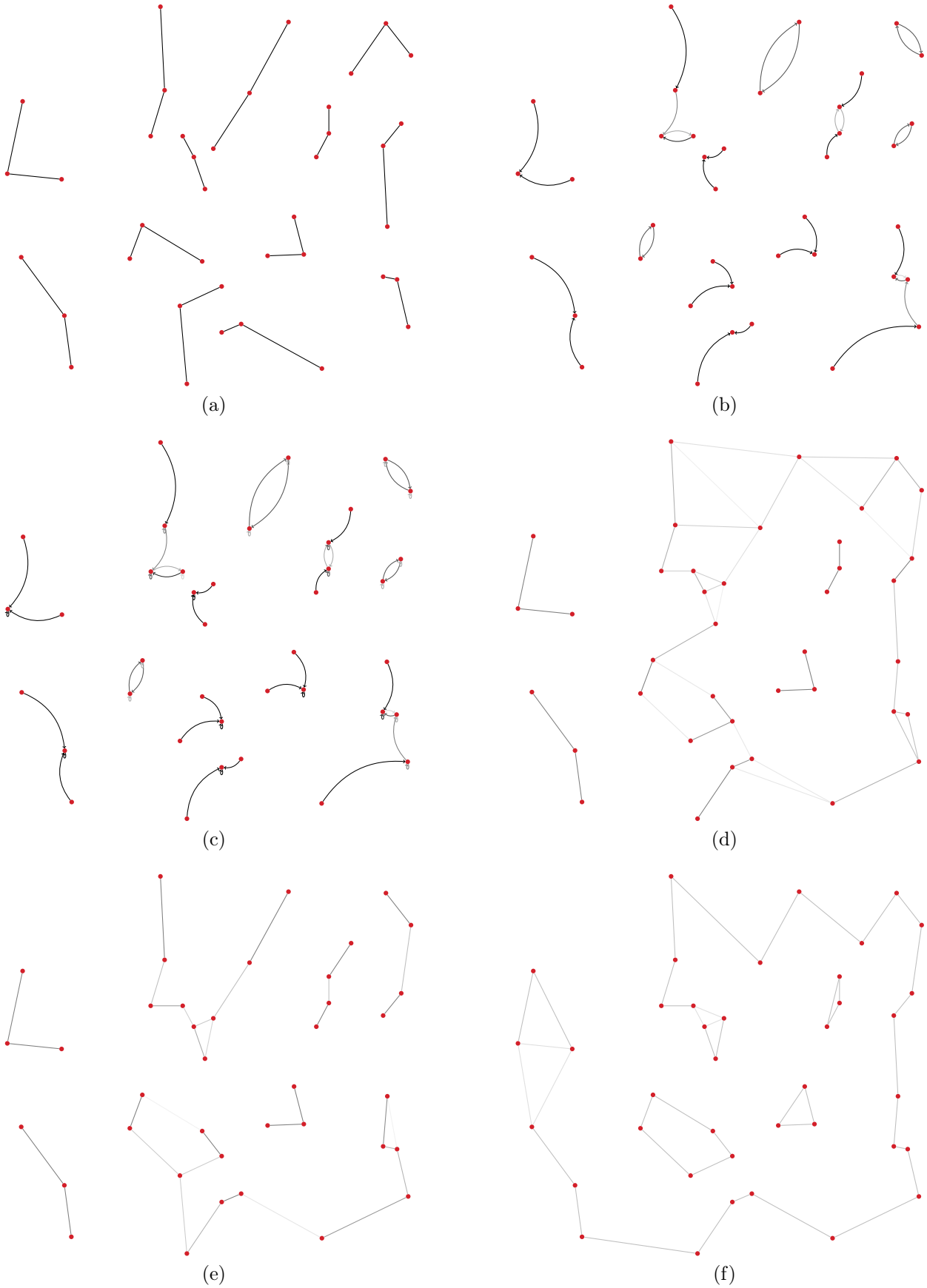
(a) (b) (c) (d) (e) (f)

Figure B.4: Instance 39a



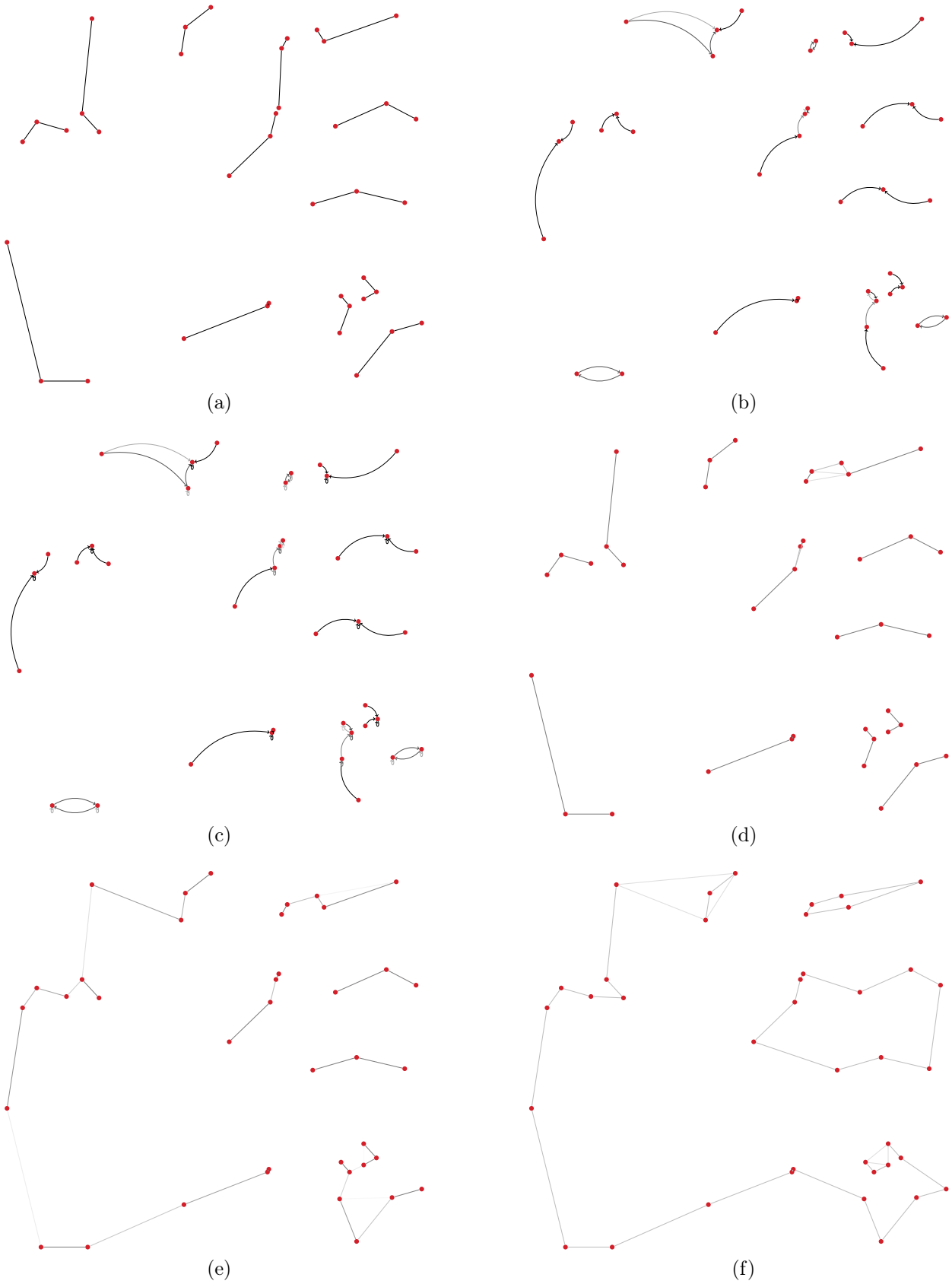
(a) (b) (c) (d) (e) (f)

Figure B.5: Instance 39b



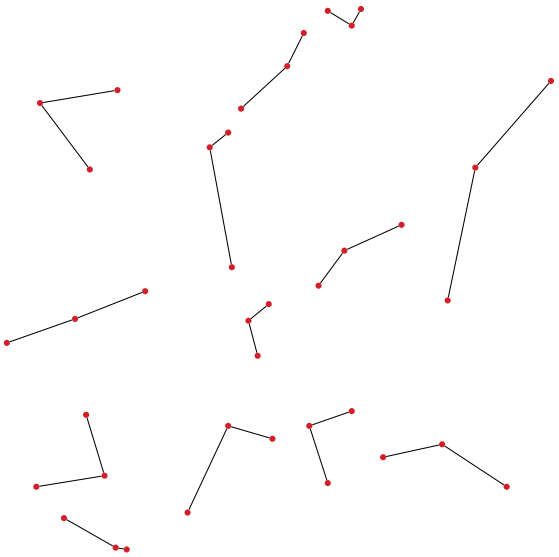
(a) (b) (c) (d) (e) (f)

Figure B.6: Instance 39c

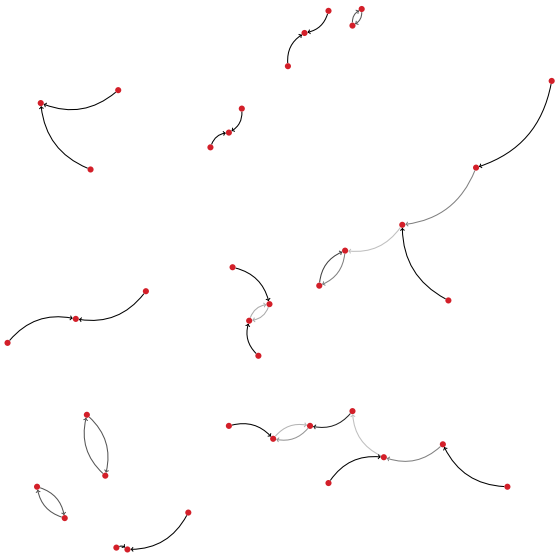


(a) (b) (c) (d) (e) (f)

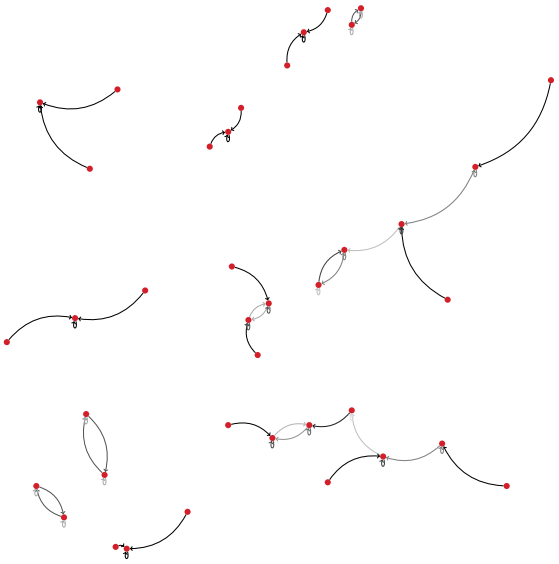
Figure B.7: Instance 39d



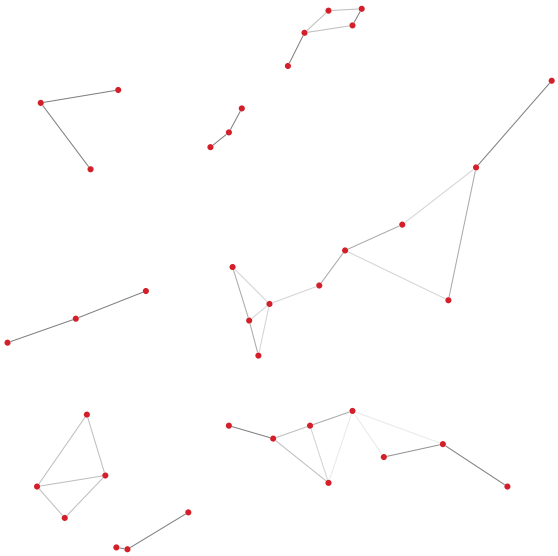
(a)



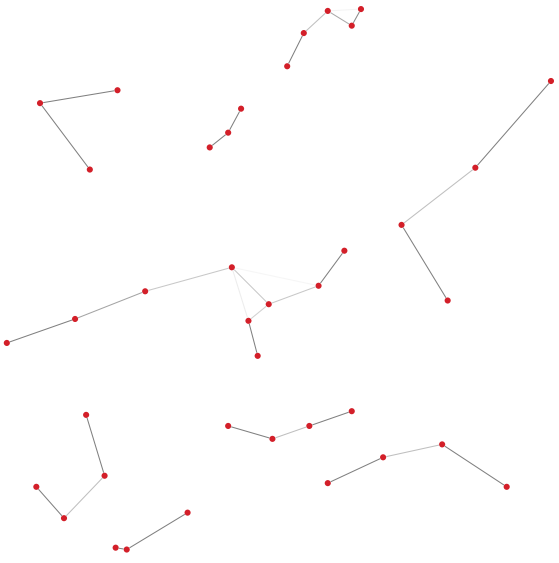
(b)



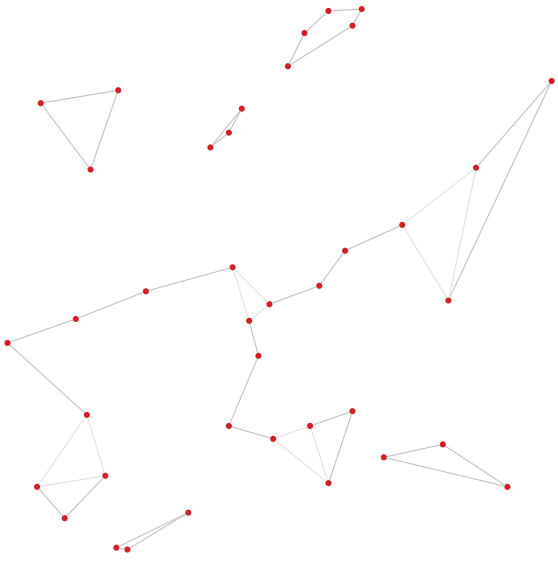
(c)



(d)

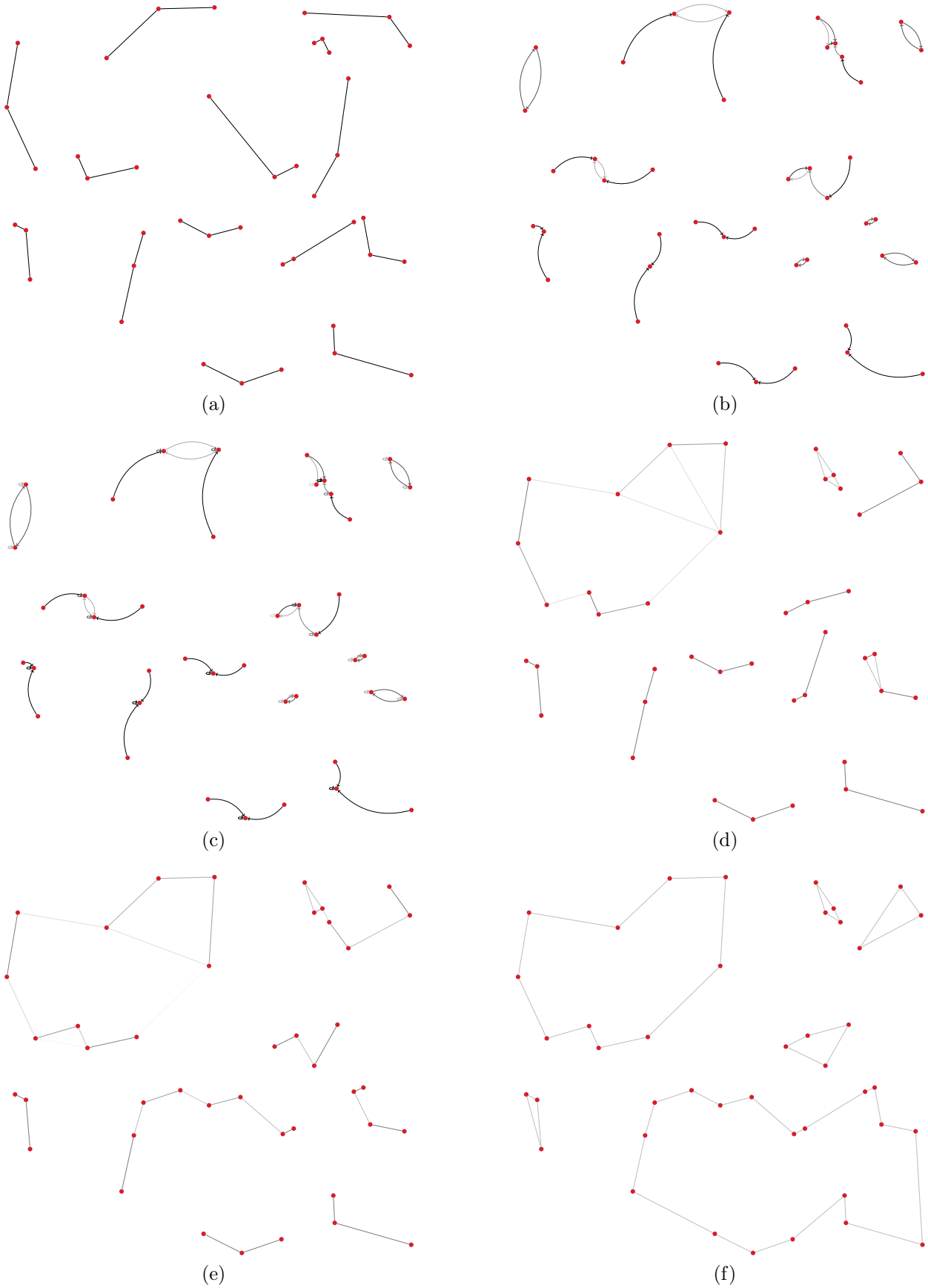


(e)



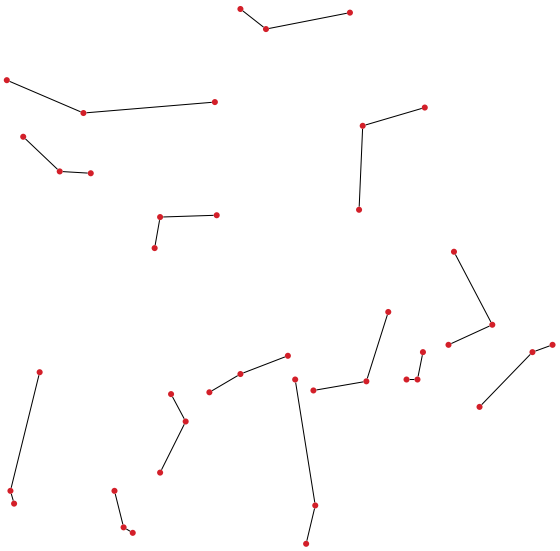
(f)

Figure B.8: Instance 39e

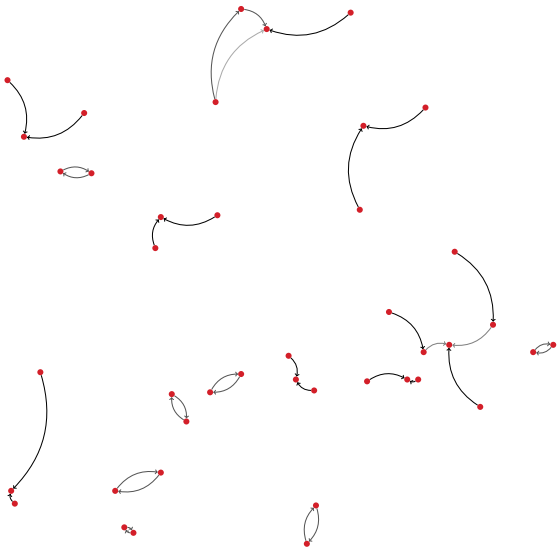


(a) (b) (c) (d) (e) (f)

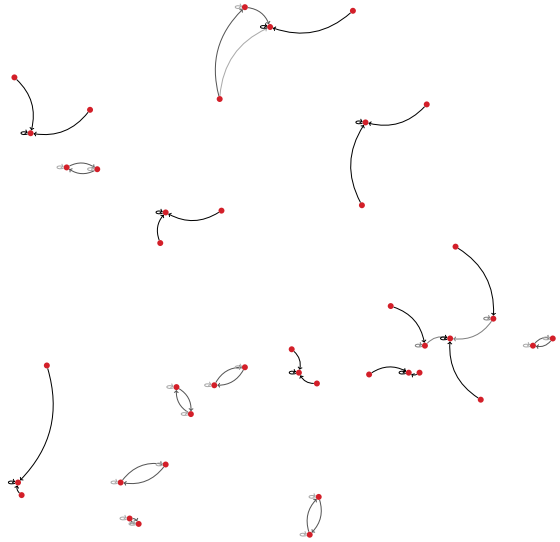
Figure B.9: Instance 42a



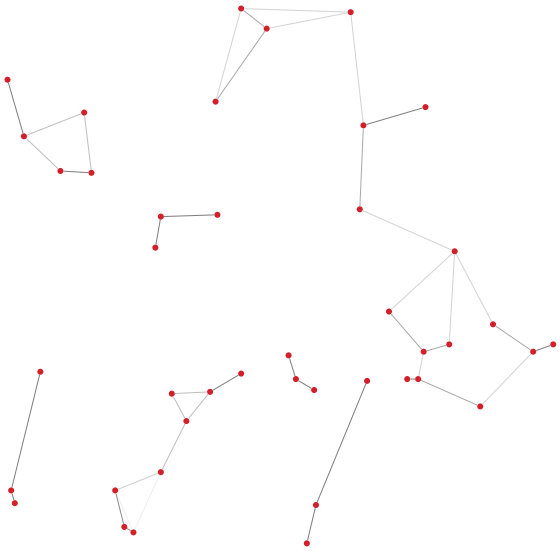
(a)



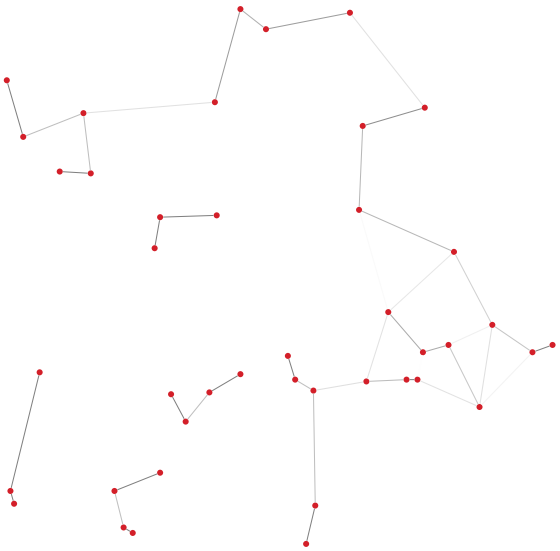
(b)



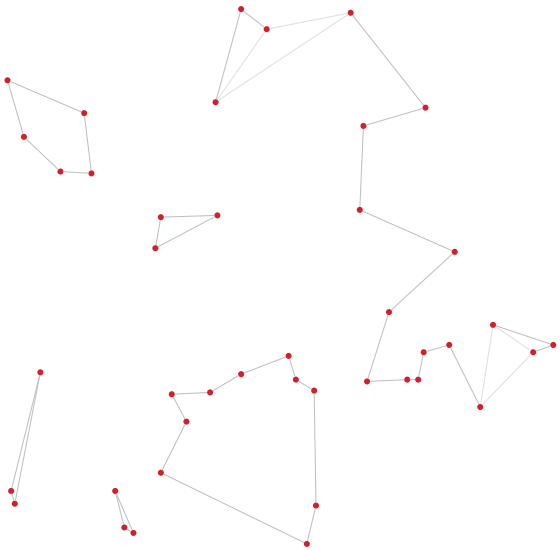
(c)



(d)

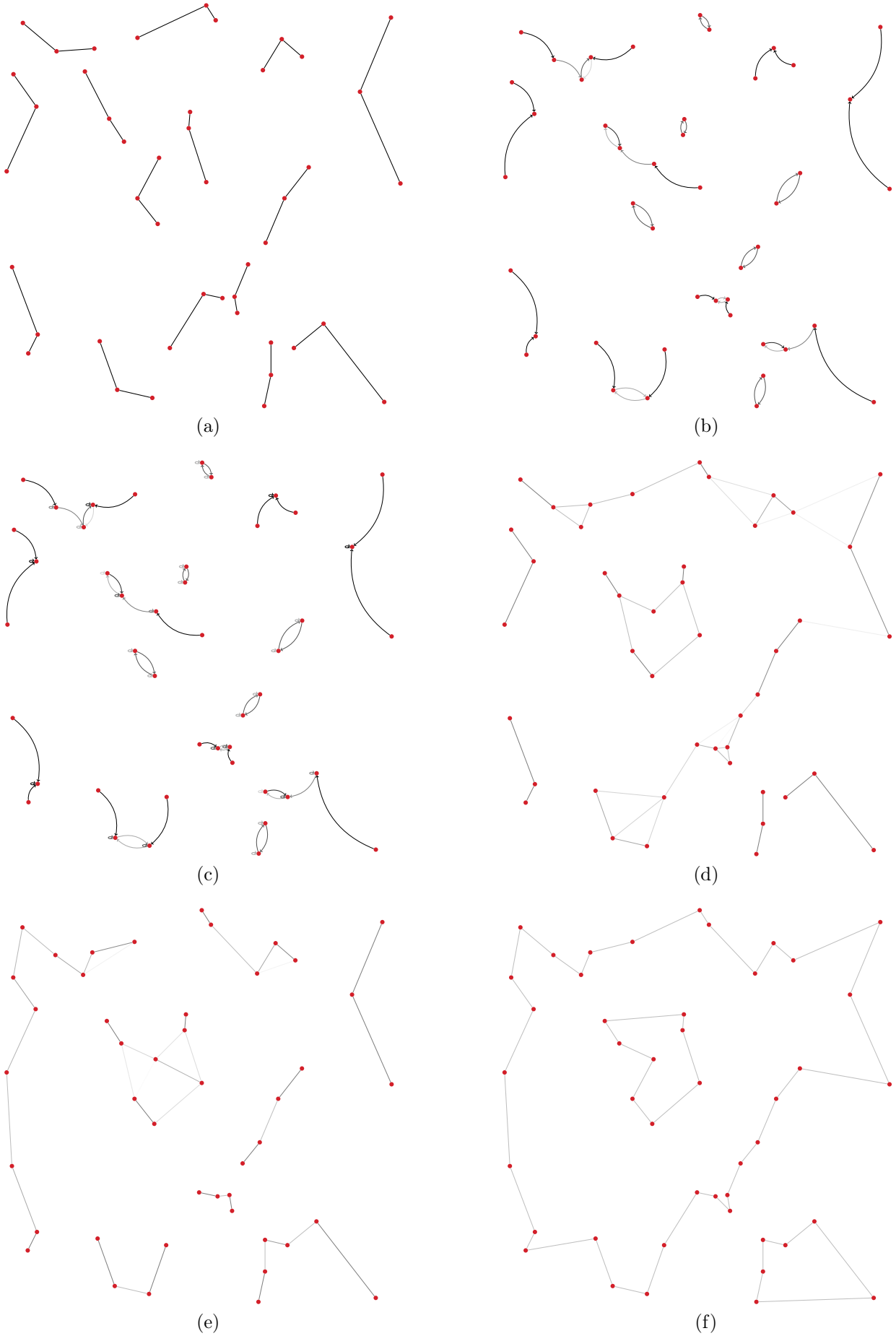


(e)



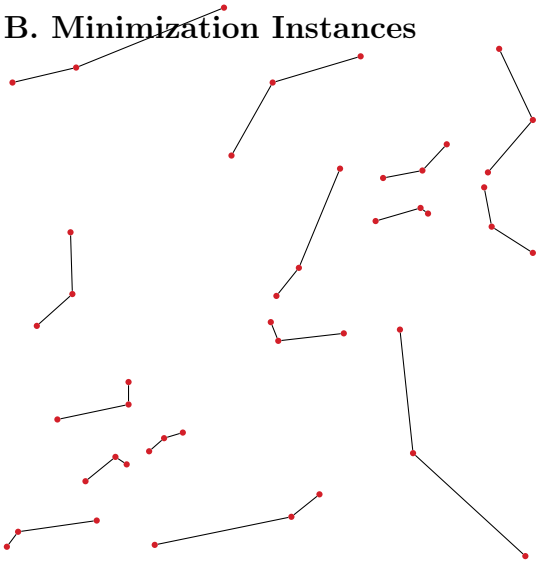
(f)

Figure B.10: Instance 42b

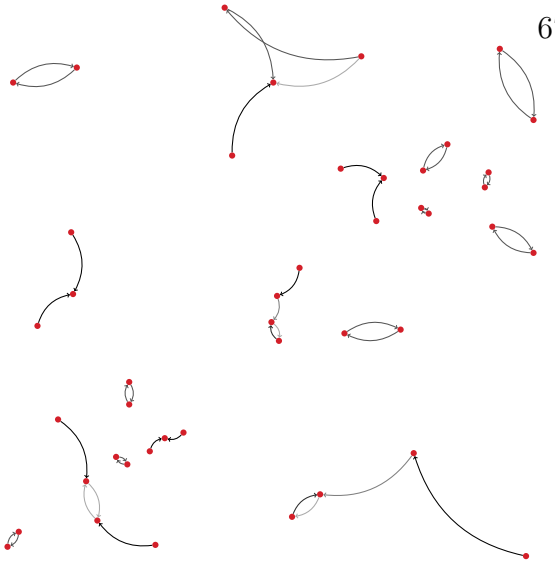


(a) (b) (c) (d) (e) (f)

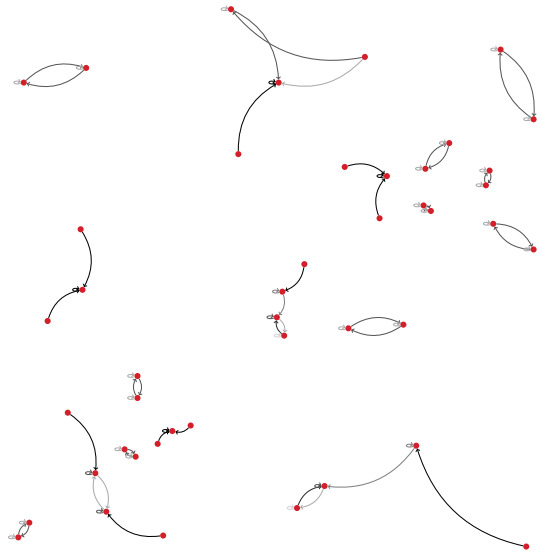
Figure B.11: Instance 45a



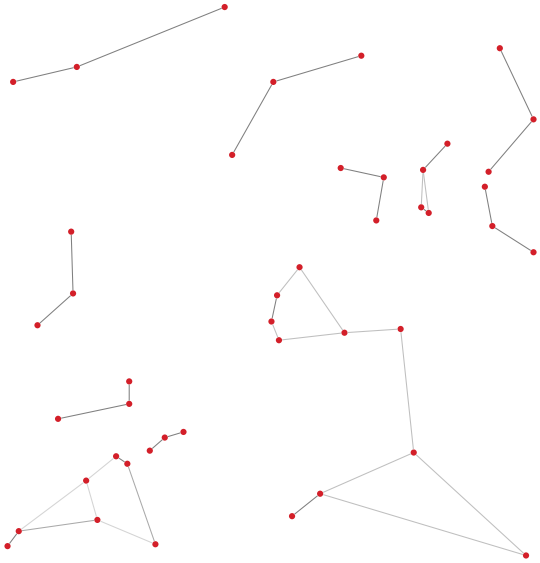
(a)



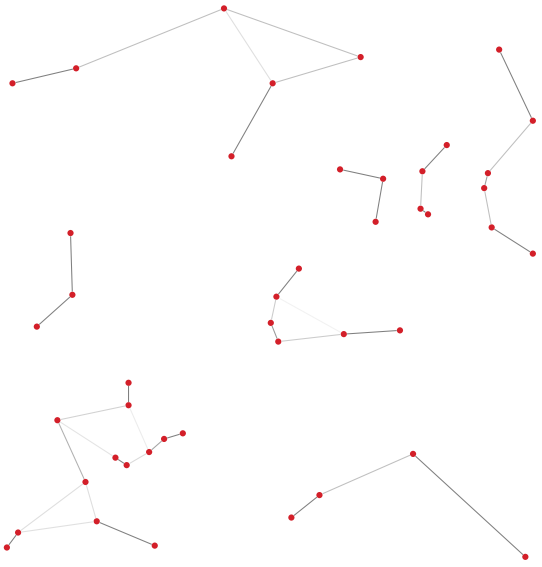
(b)



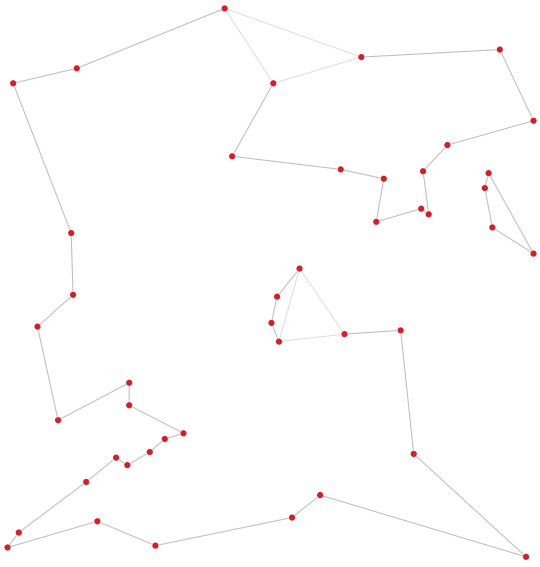
(c)



(d)

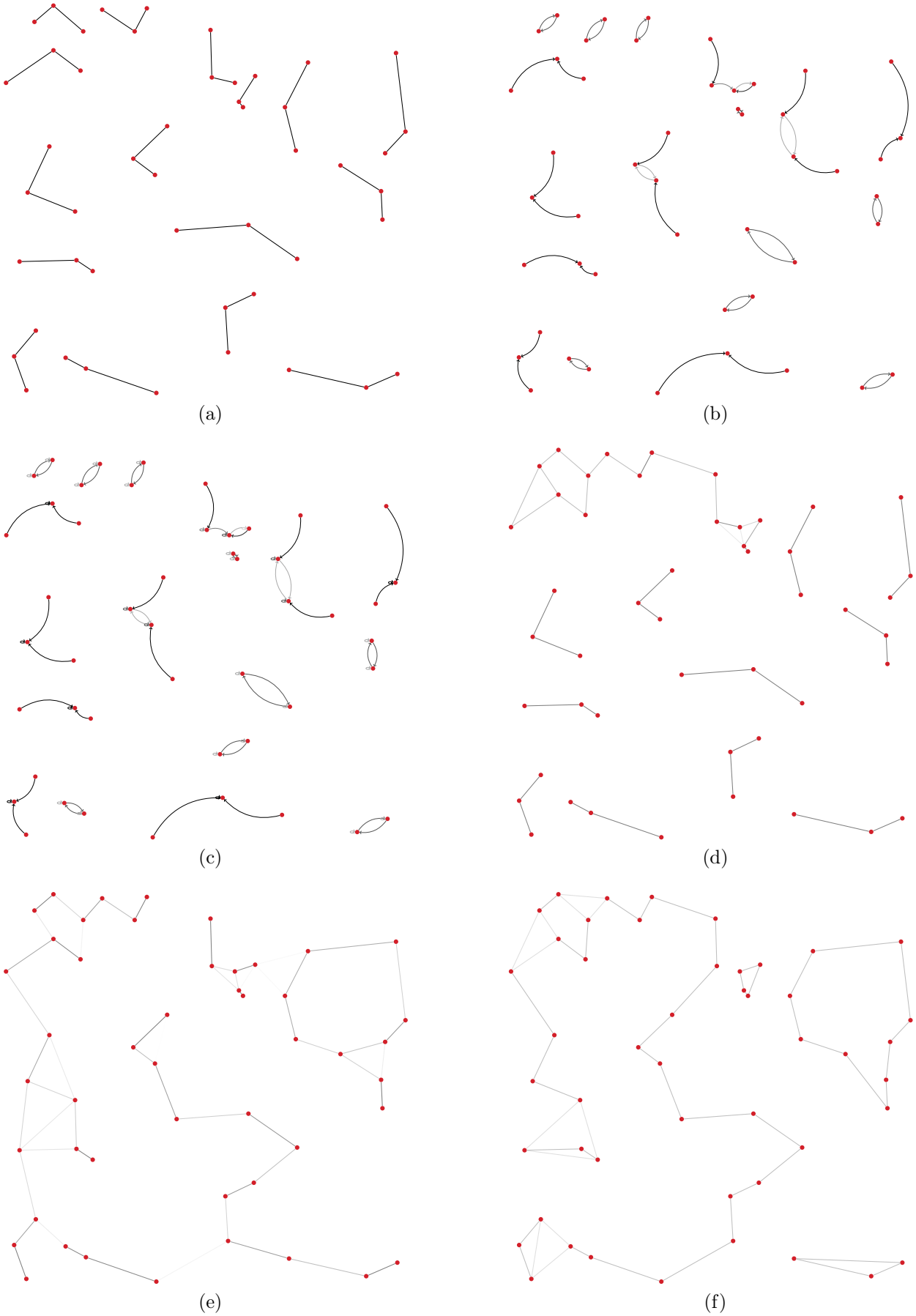


(e)



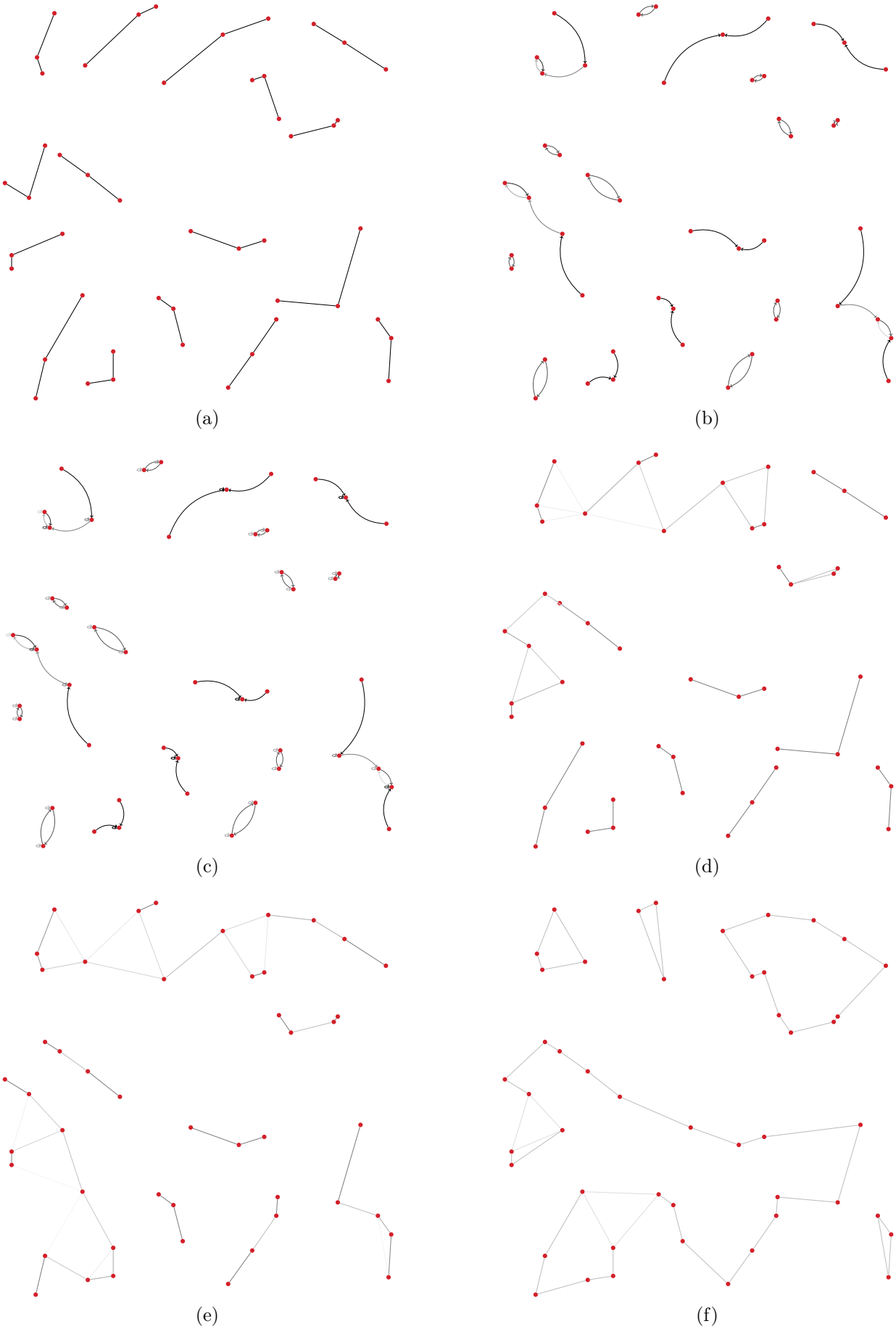
(f)

Figure B.12: Instance 45b



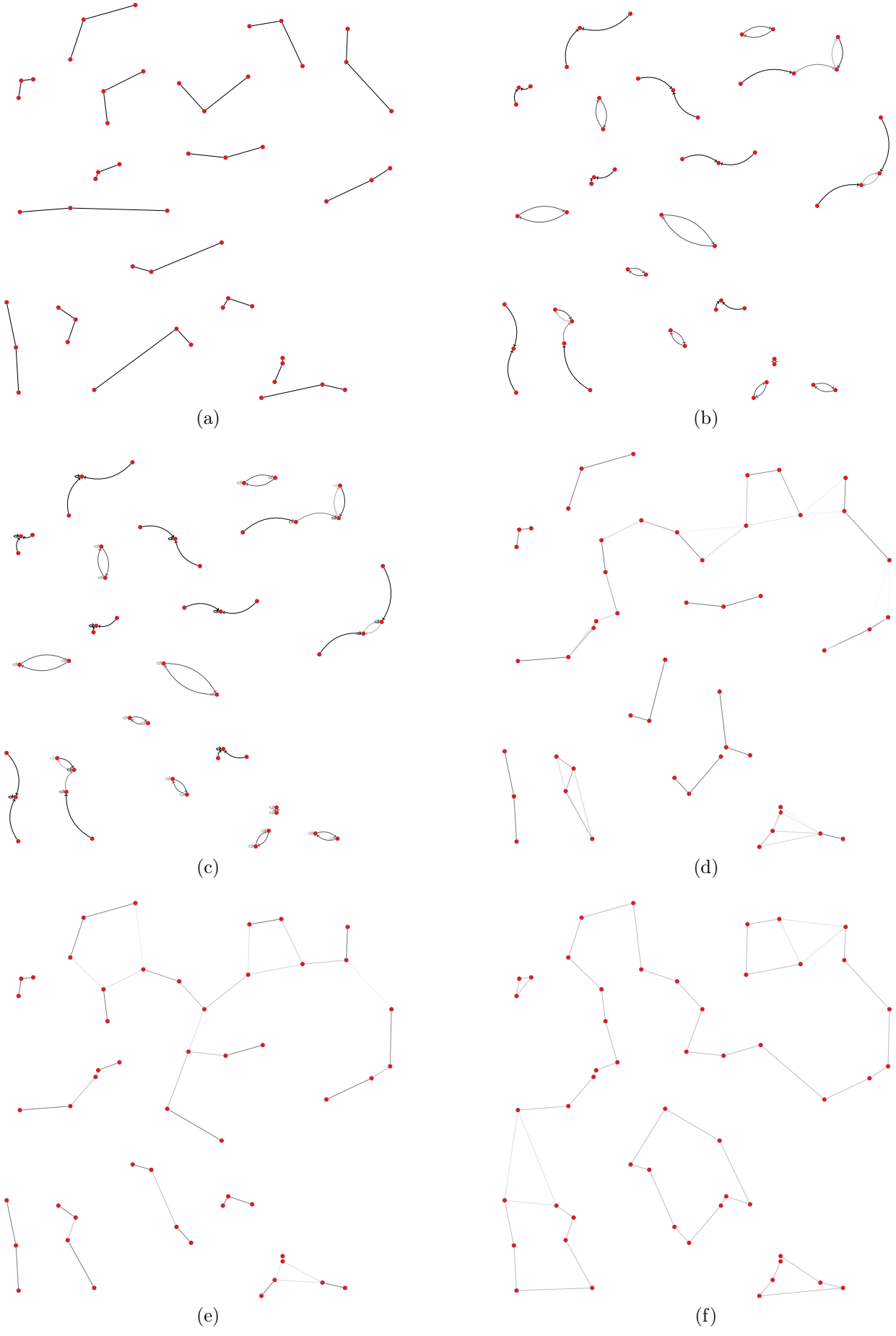
(a) (b) (c) (d) (e) (f)

Figure B.13: Instance 48a



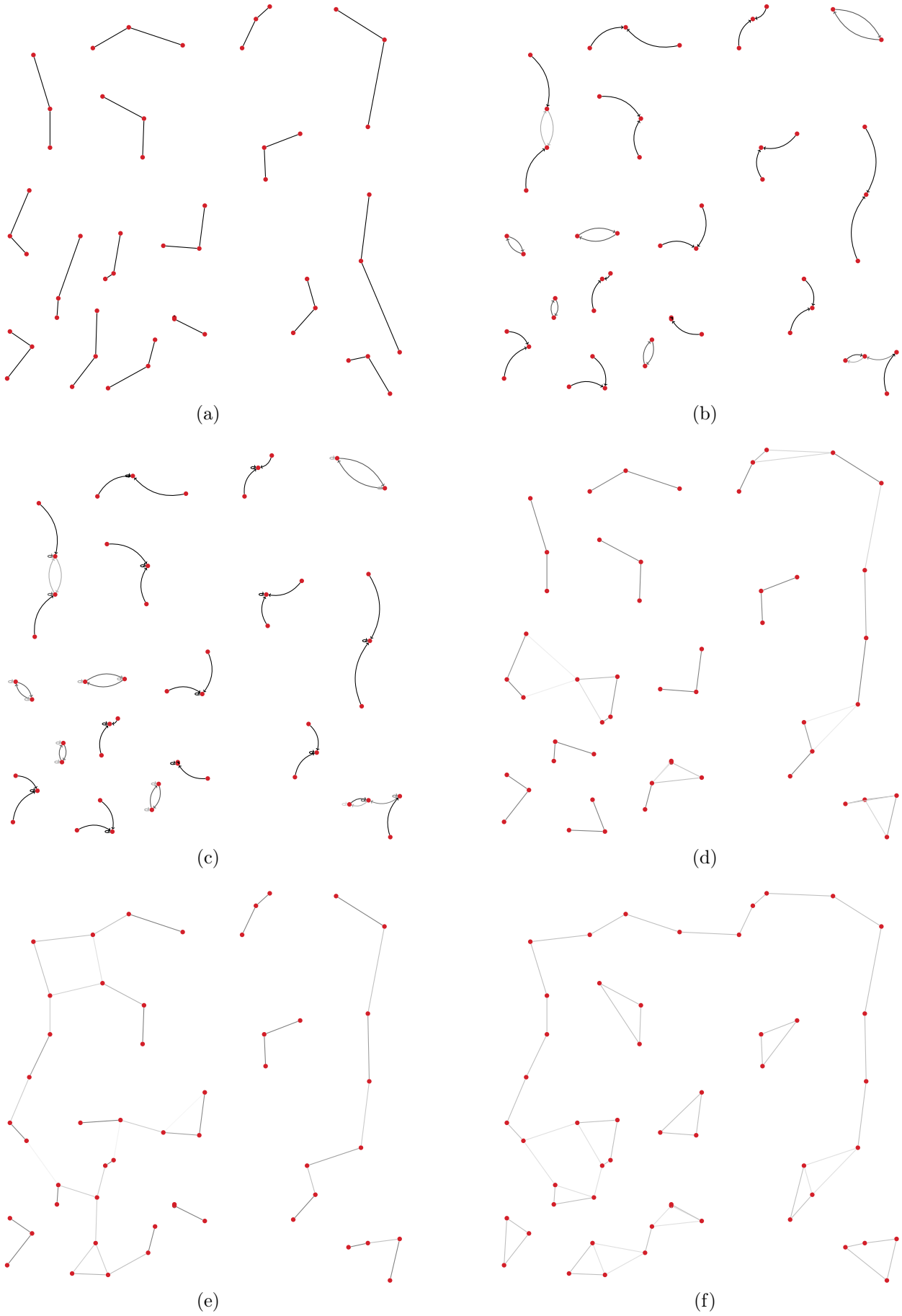
(a) (b) (c) (d) (e) (f)

Figure B.14: Instance 48a



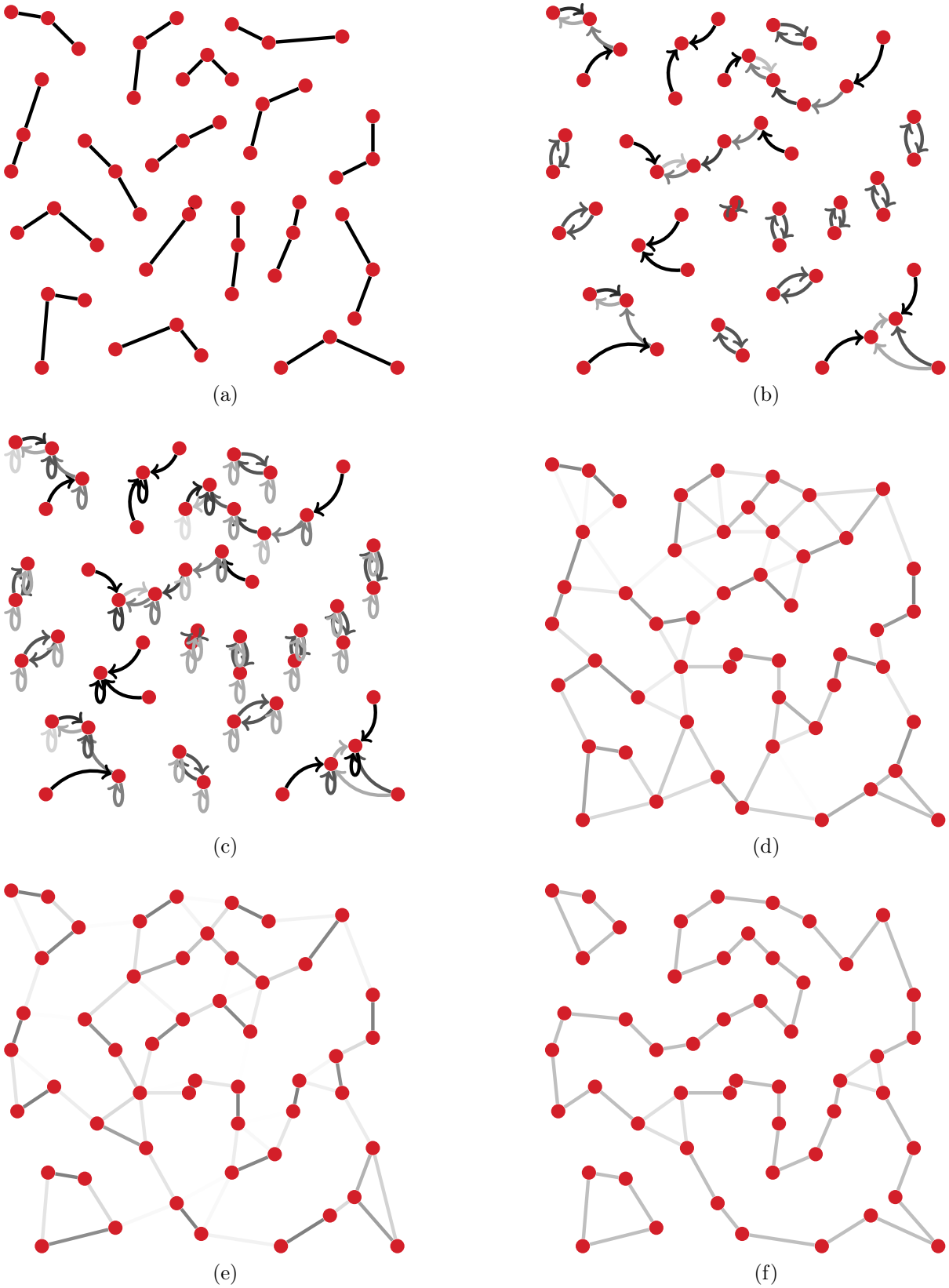
(a) (b) (c) (d) (e) (f)

Figure B.15: Instance 51a



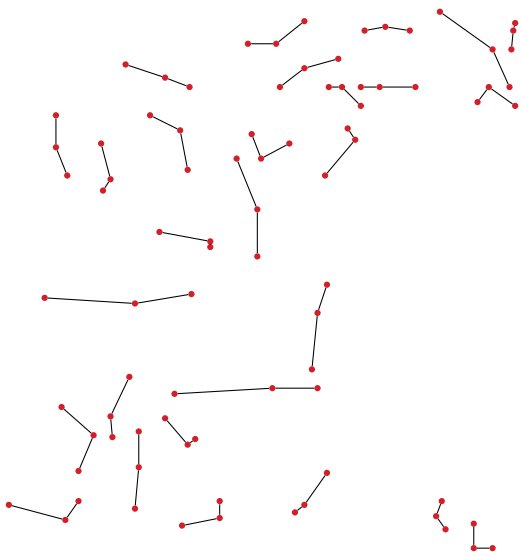
(a) (b) (c) (d) (e) (f)

Figure B.16: Instance 51b

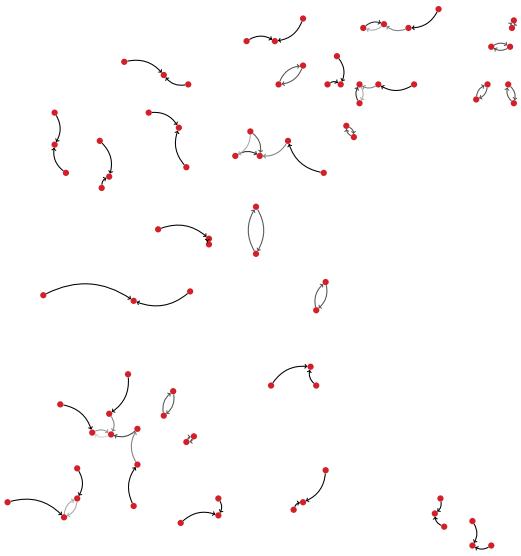


(a) (b) (c) (d) (e) (f)

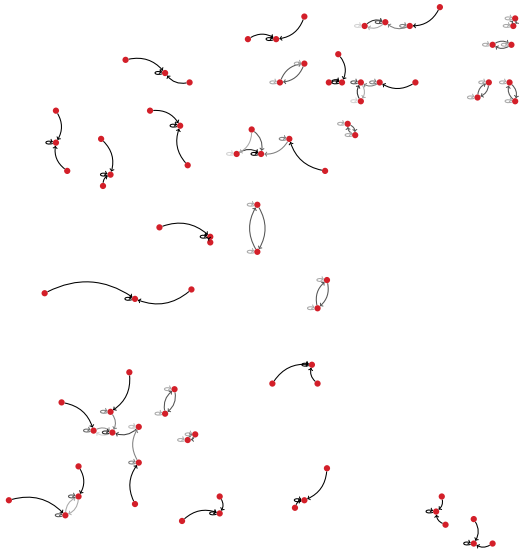
Figure B.17: Instance h01



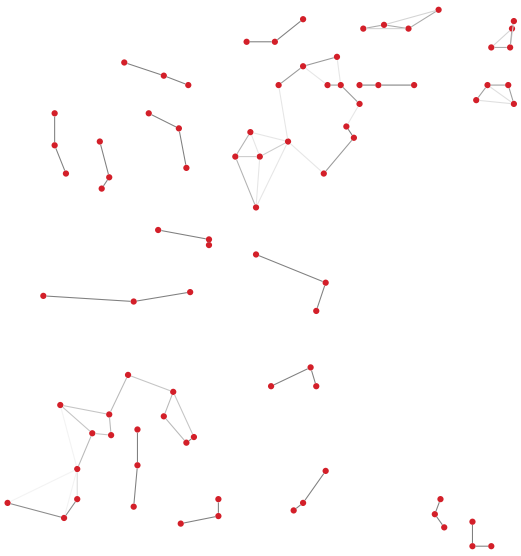
(a)



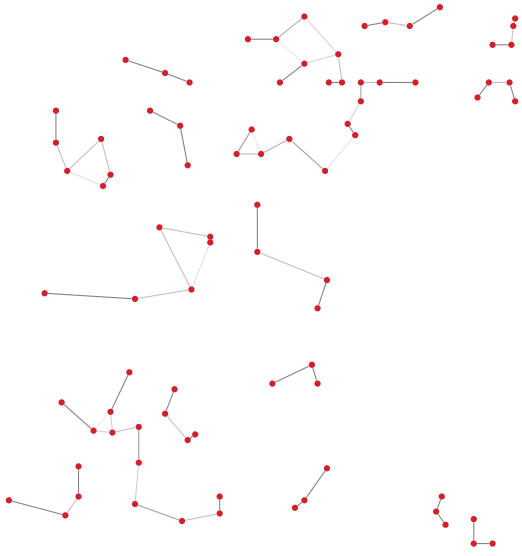
(b)



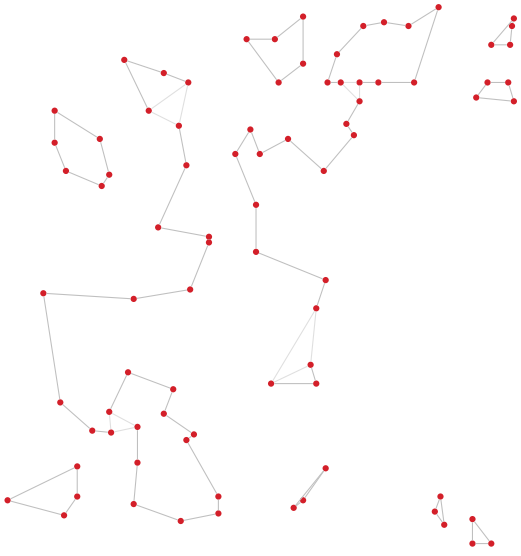
(c)



(d)

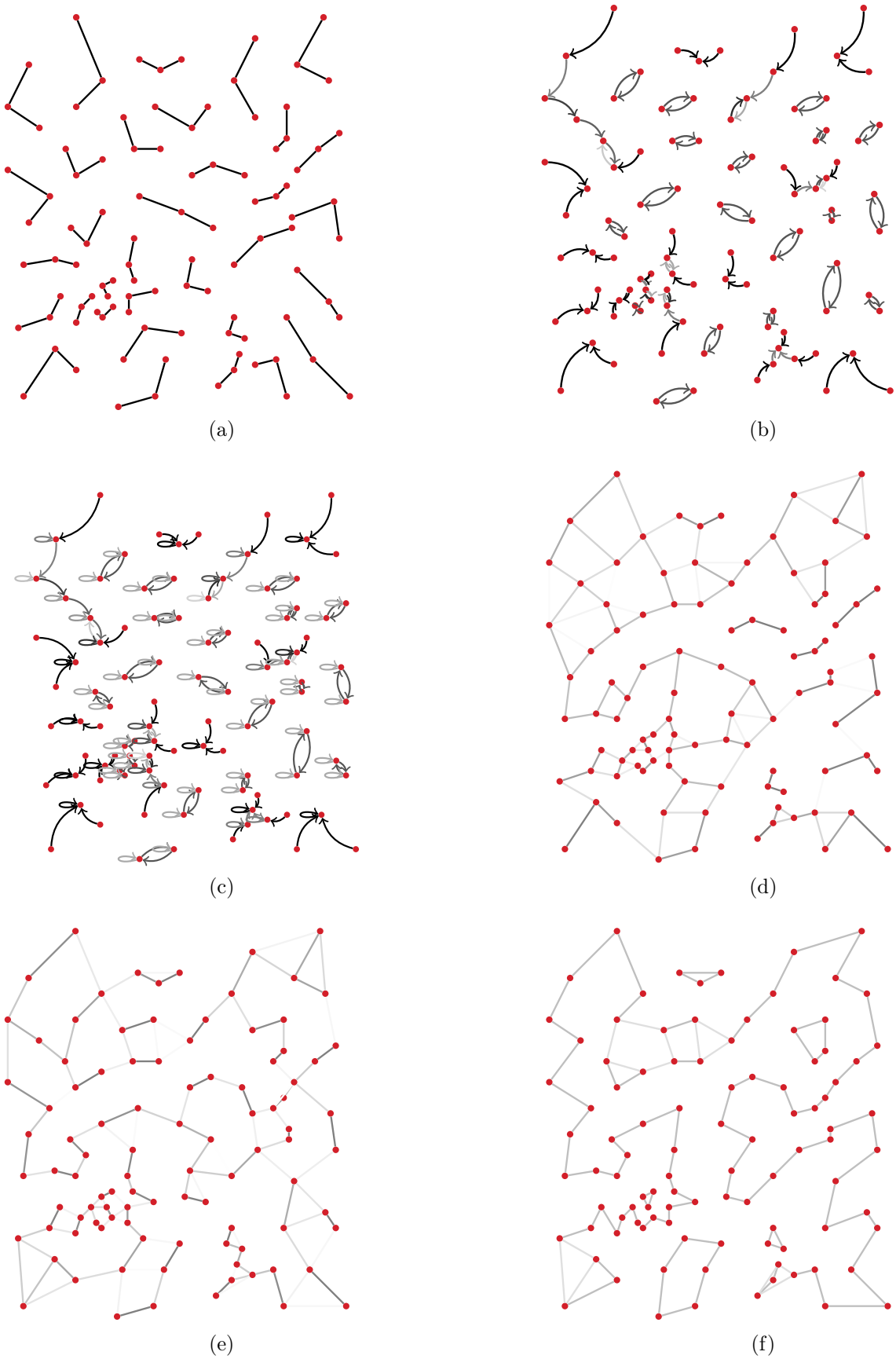


(e)



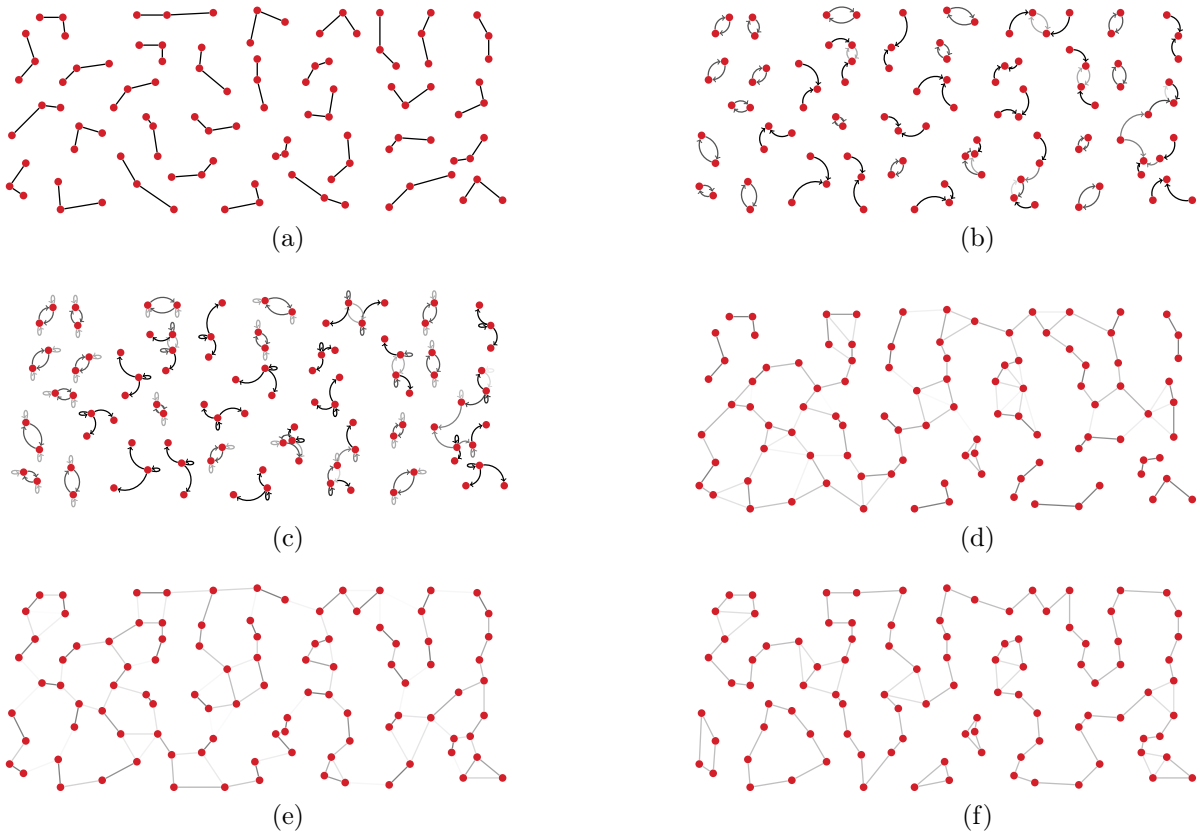
(f)

Figure B.18: Instance h02



(a) (b) (c) (d) (e) (f)

Figure B.19: Instance f99



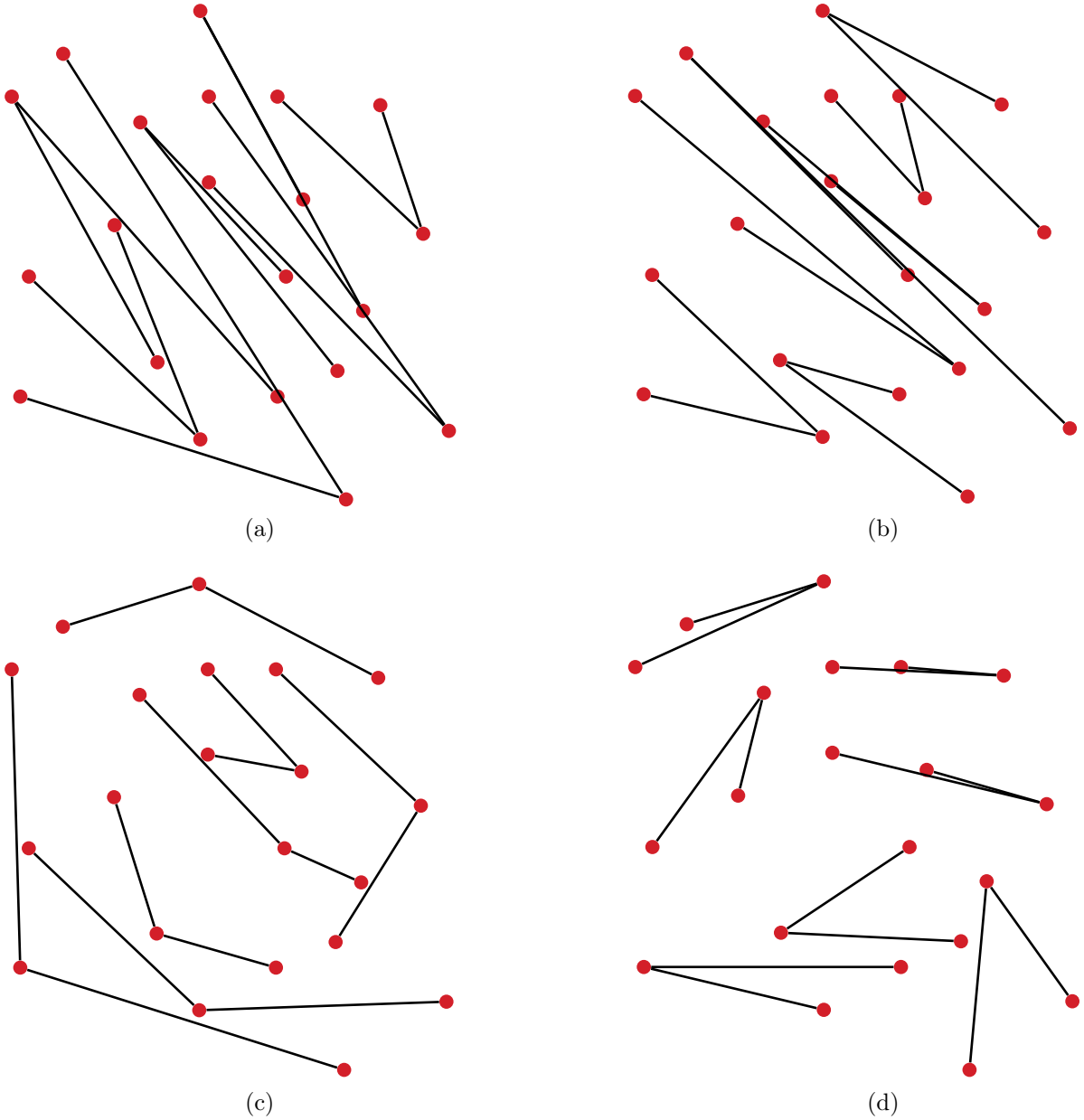
(a) (b) (c) (d) (e) (f)

Figure B.20: Instance h03

Appendix C

Maximization instances

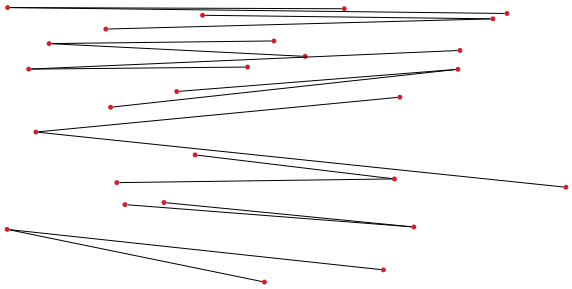
In this appendix, we draw the benchmark instances used for the maximization version of the 3-matching problem without crossings. We list the (a) optimal solution, (b) solution found by the Windose heuristic, (c) solution found by the ConvHull heuristic and (d) solution found by the Guillotine heuristic.



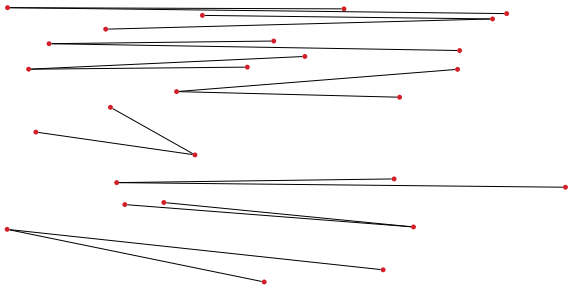
(a) (b)

(c) (d)

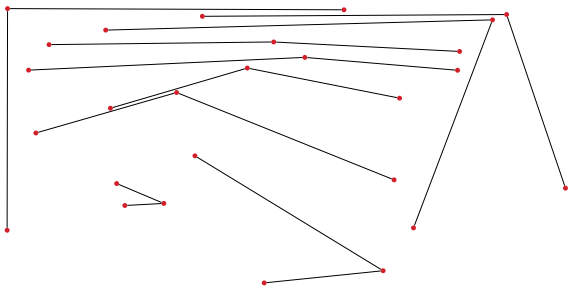
Figure C.1: Instance f21



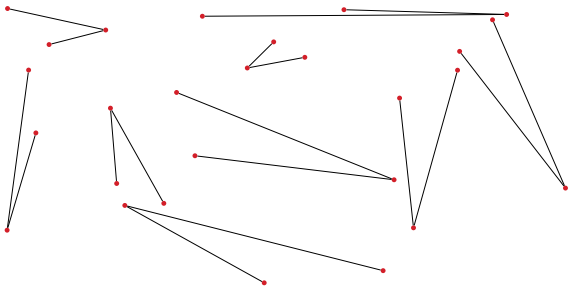
(a)



(b)

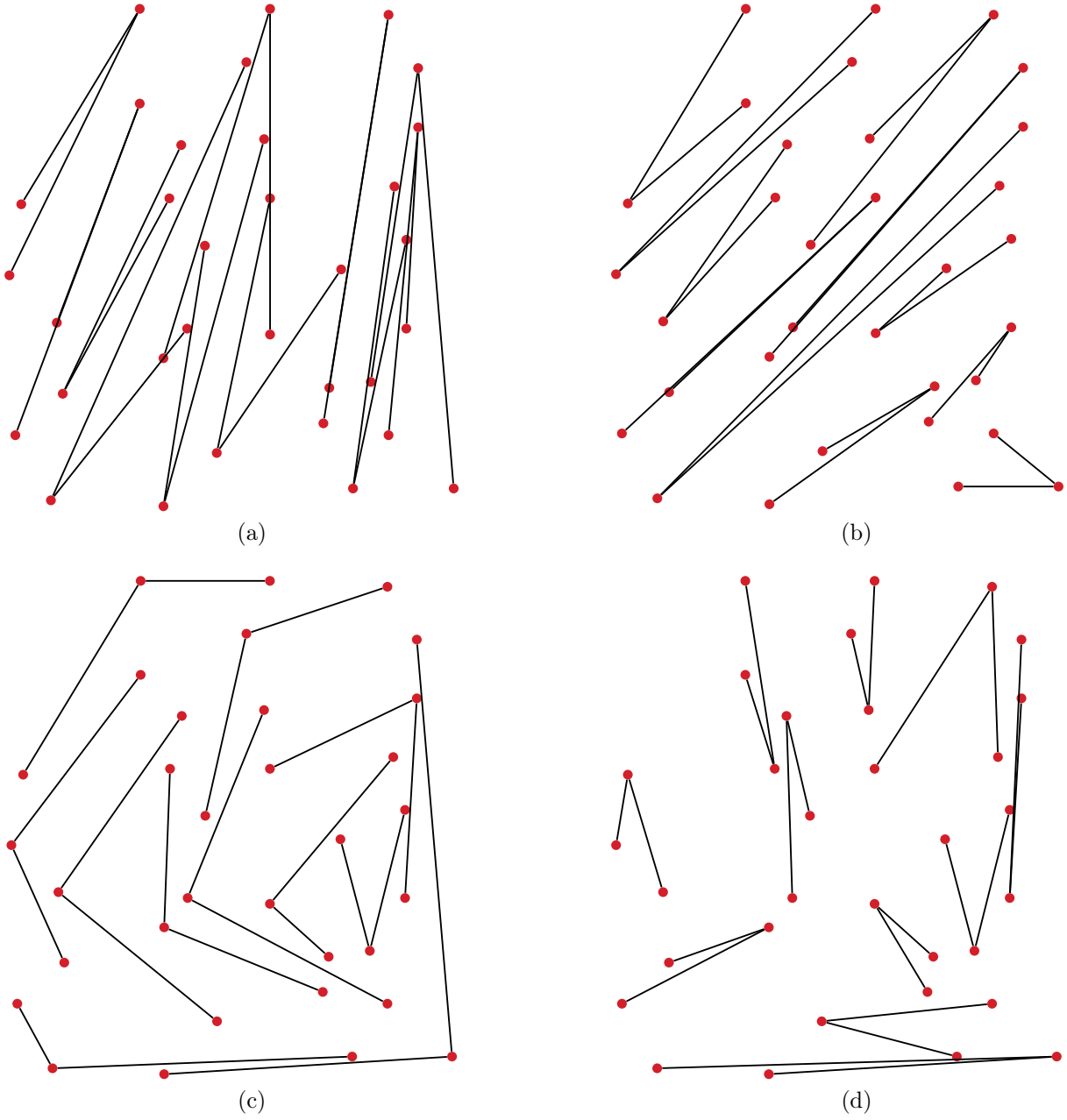


(c)



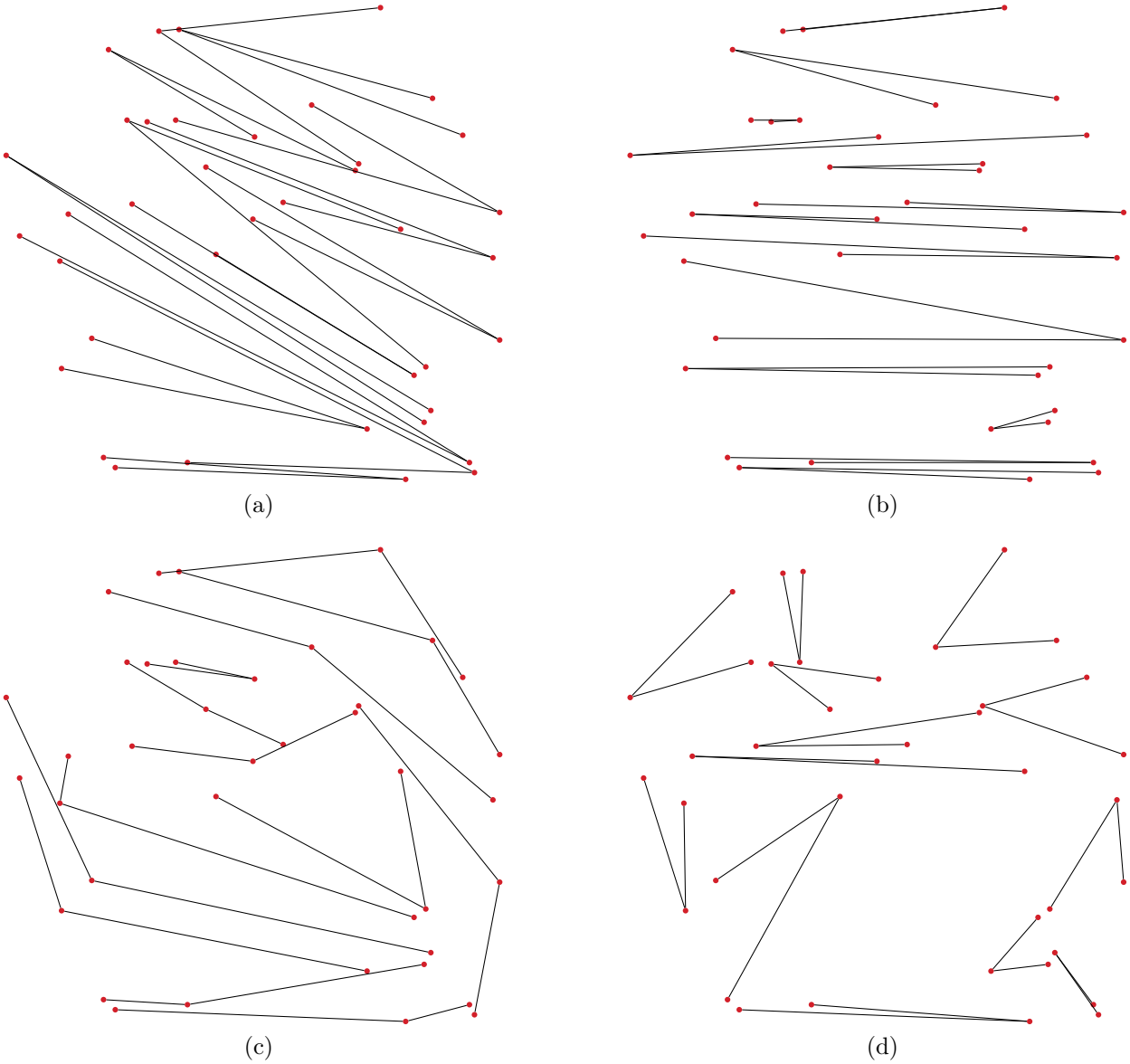
(d)

Figure C.2: Instance f27



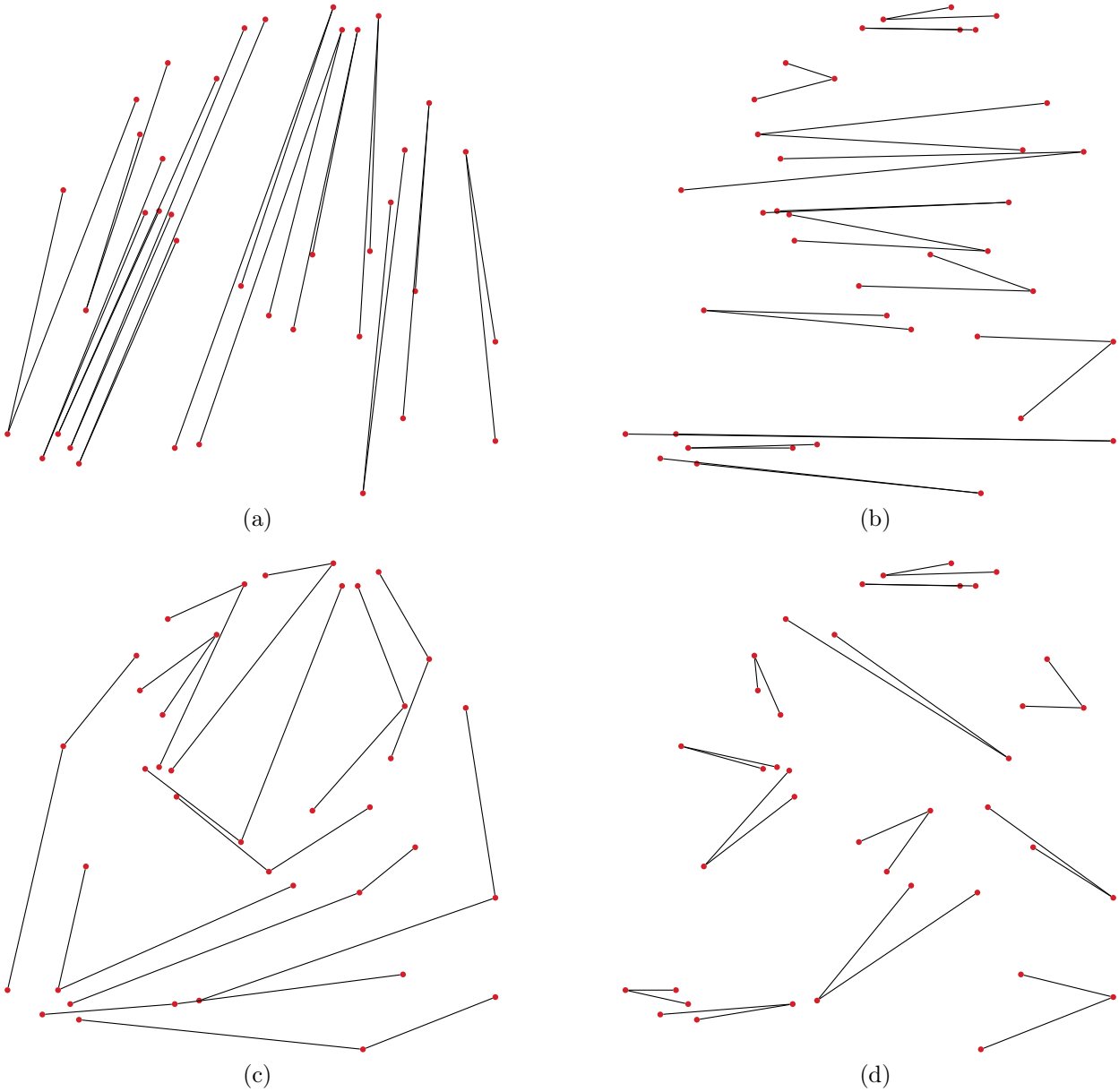
(a) (b) (c) (d)

Figure C.3: Instance f33



(a) (b) (c) (d)

Figure C.4: Instance 39a



(a) (b) (c) (d)

Figure C.5: Instance 39b

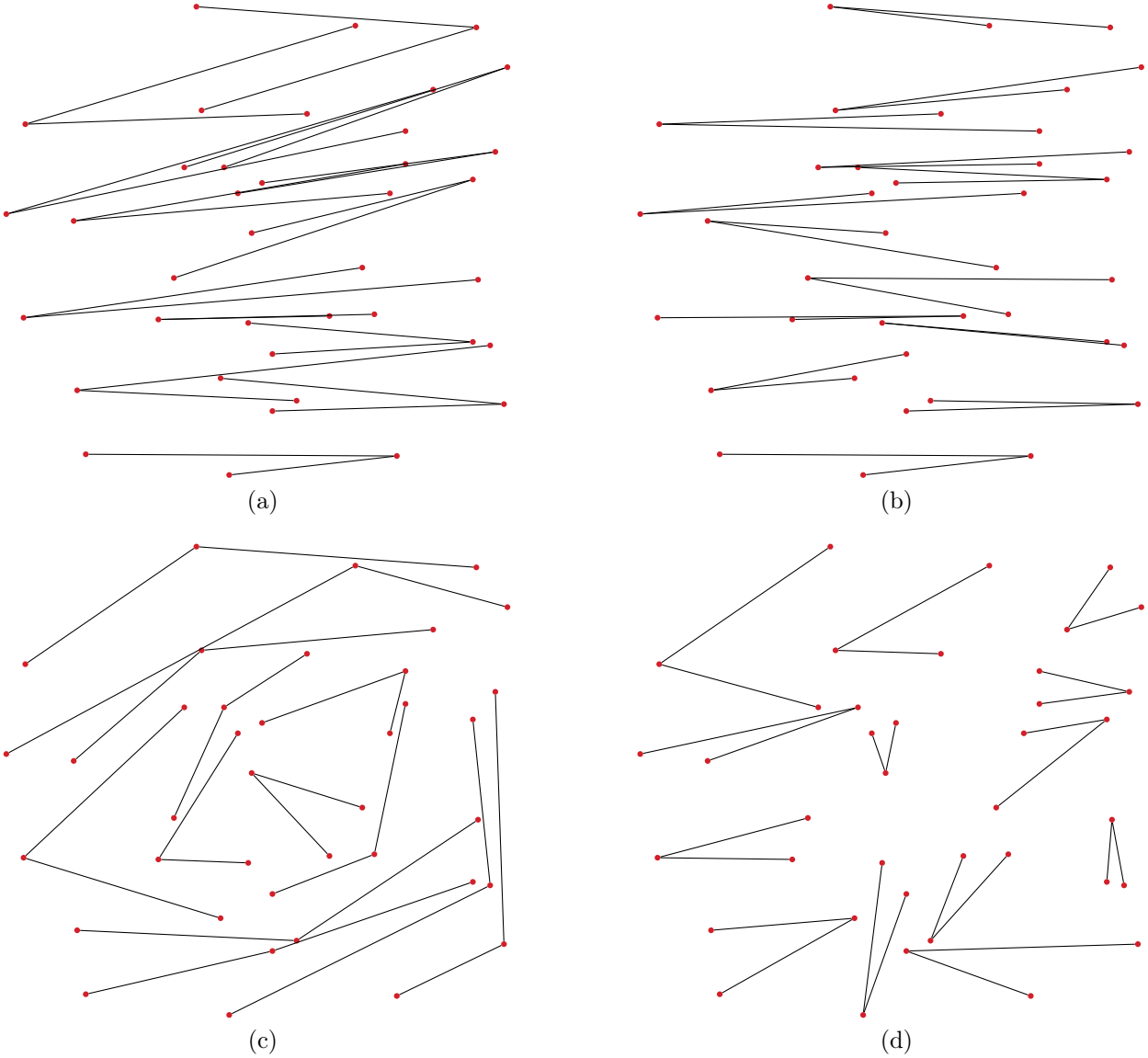
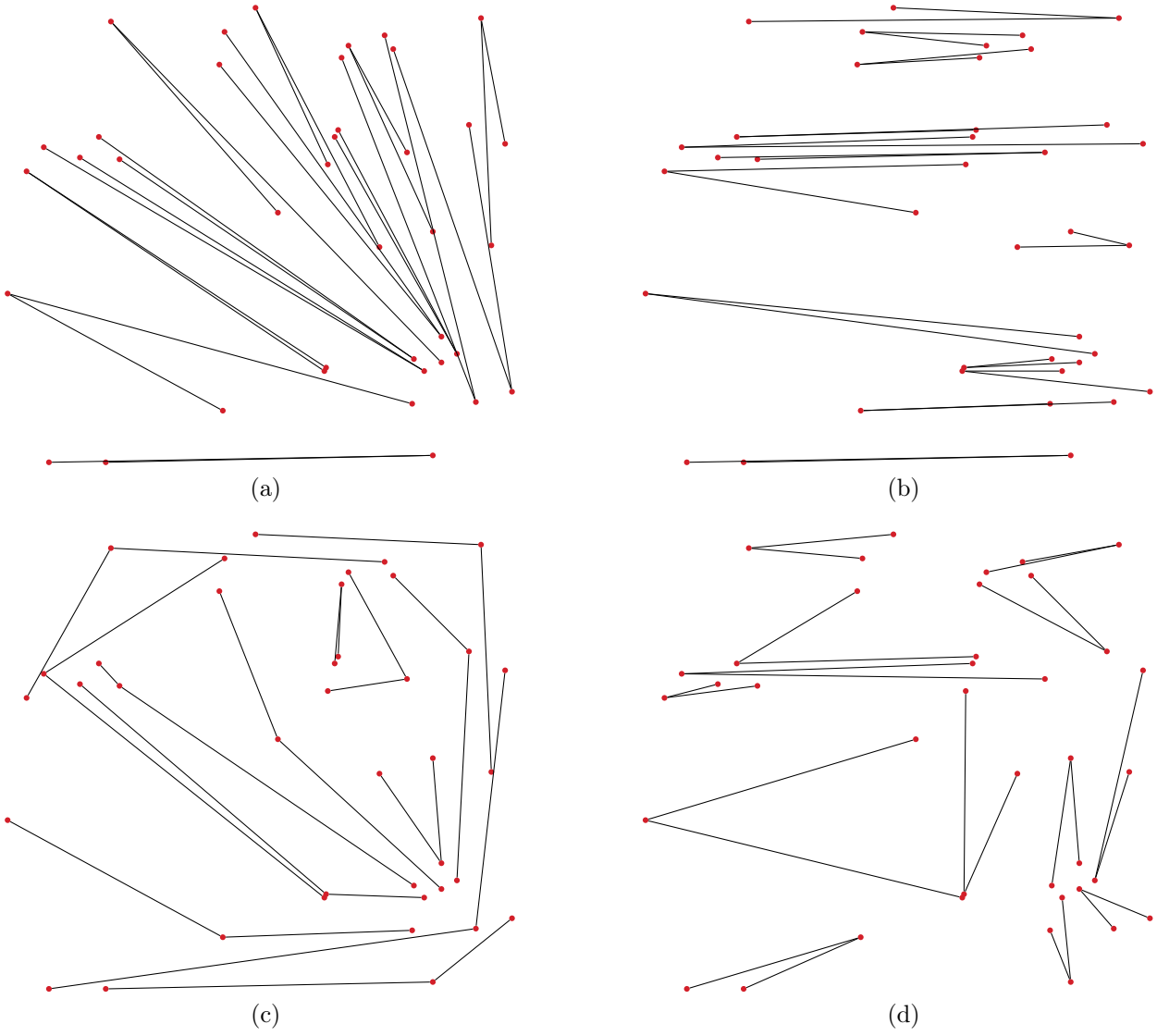
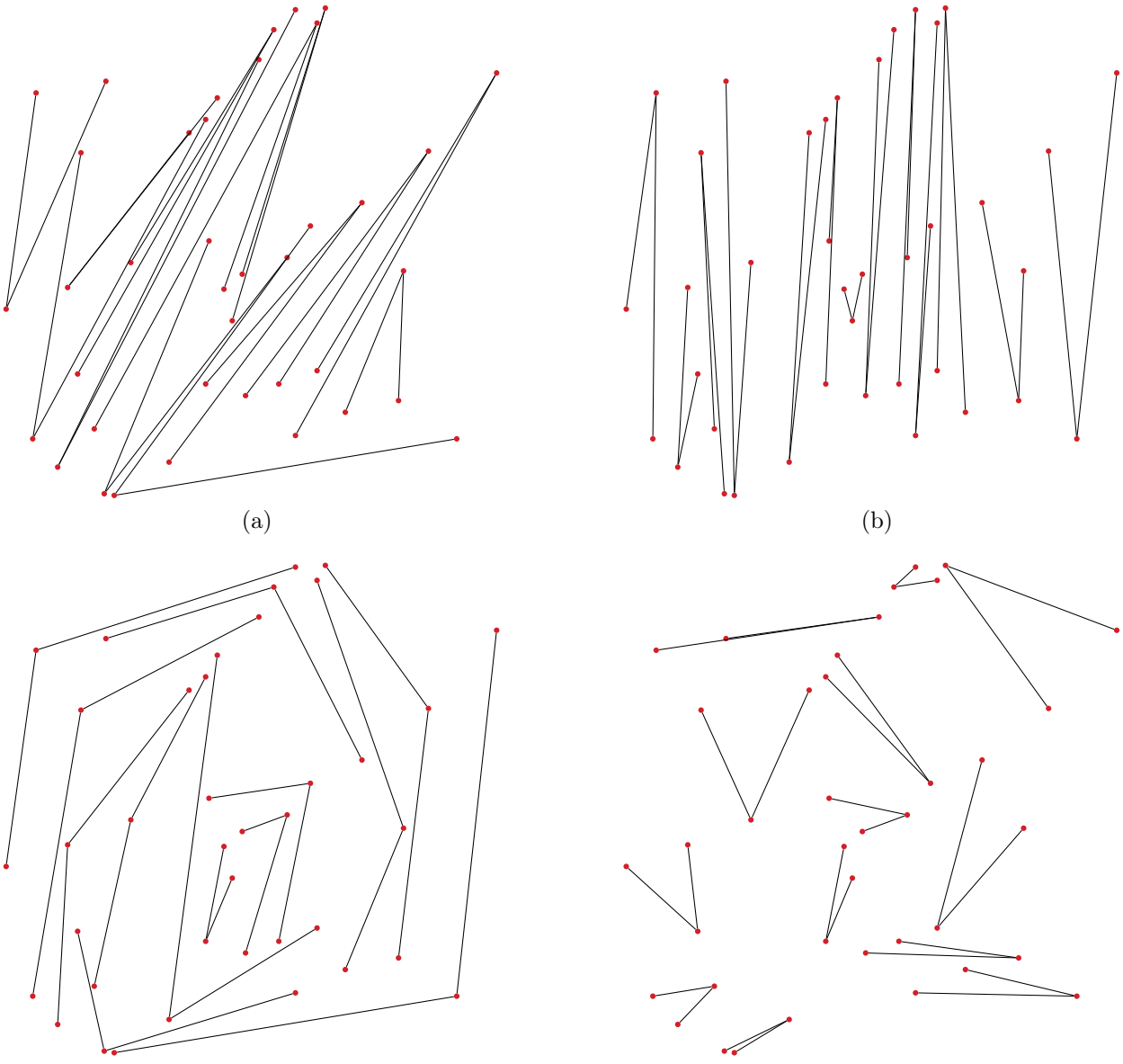


Figure C.6: Instance 39c



(a) (b) (c) (d)

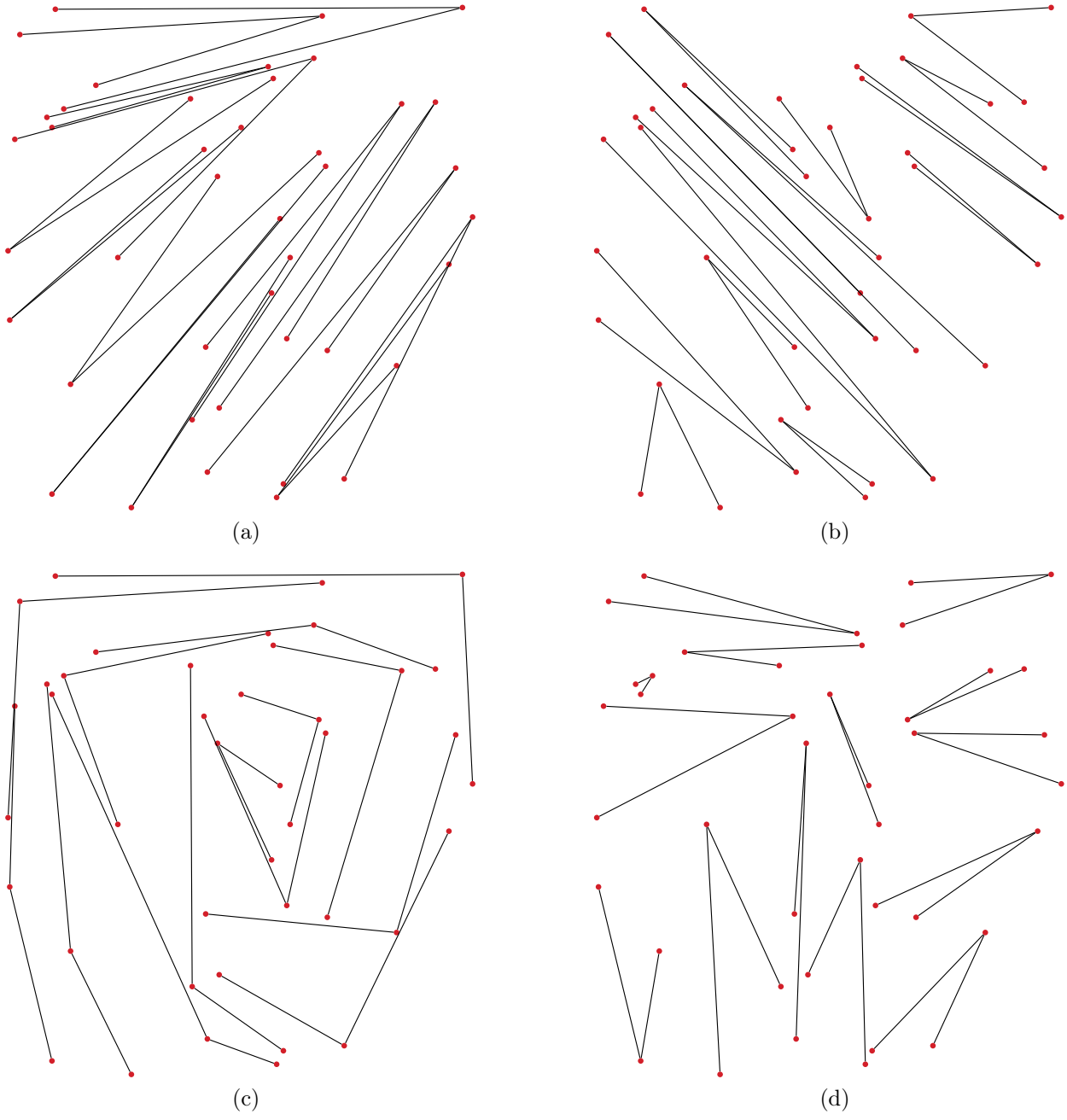
Figure C.7: Instance 39d



(a) (b)

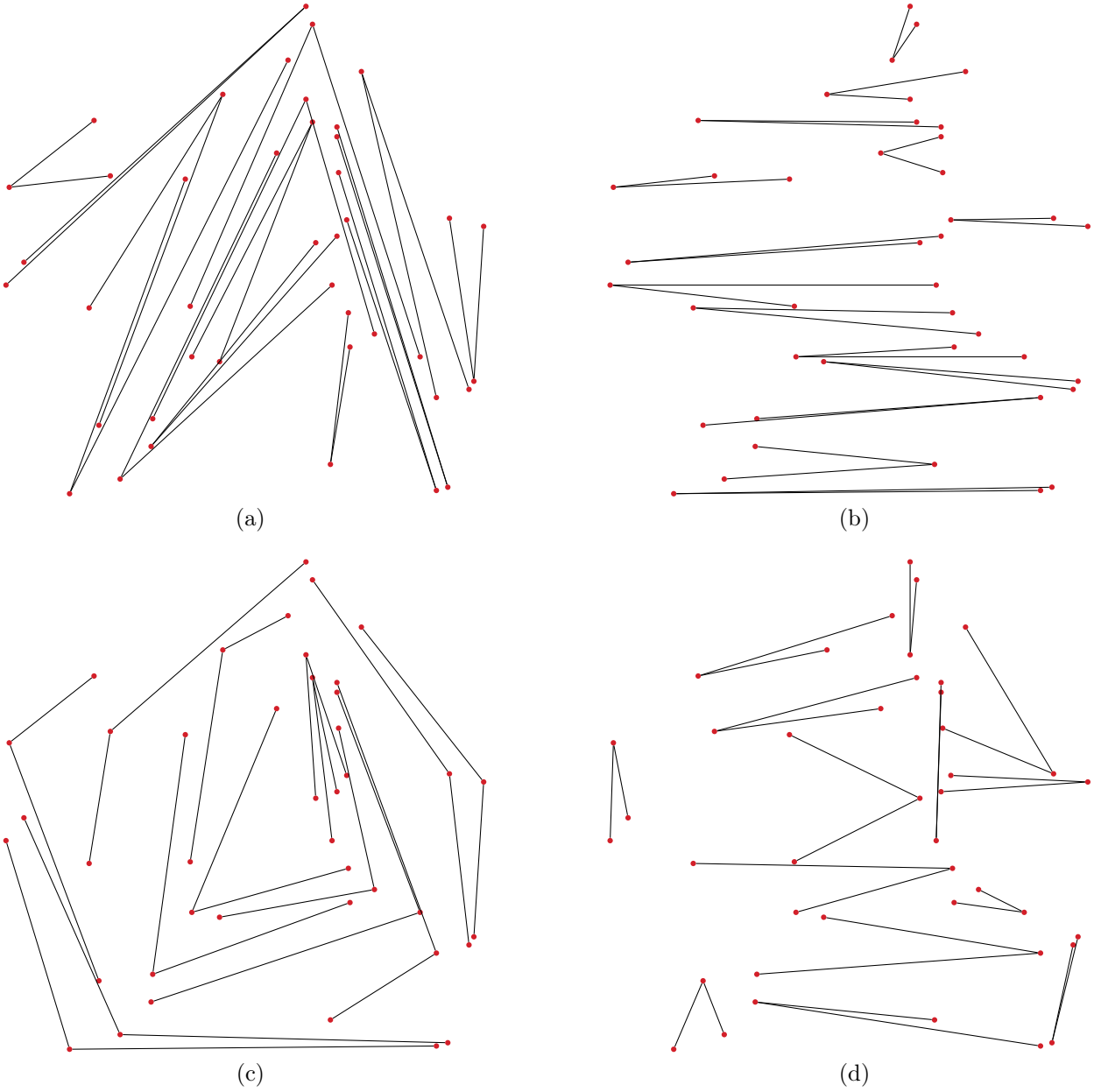
(c) (d)

Figure C.8: Instance 39e



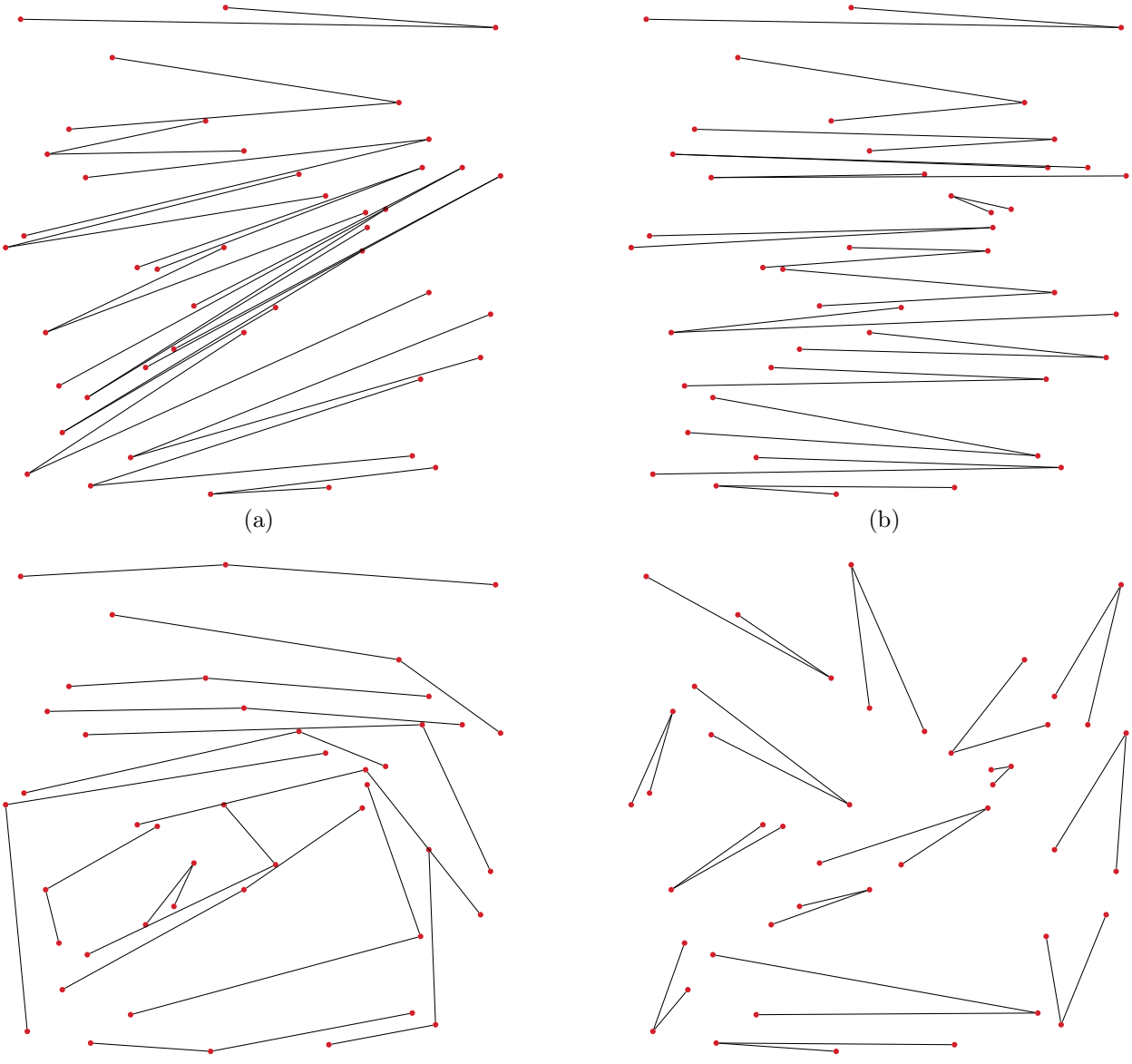
(a) (b) (c) (d)

Figure C.9: Instance 42a



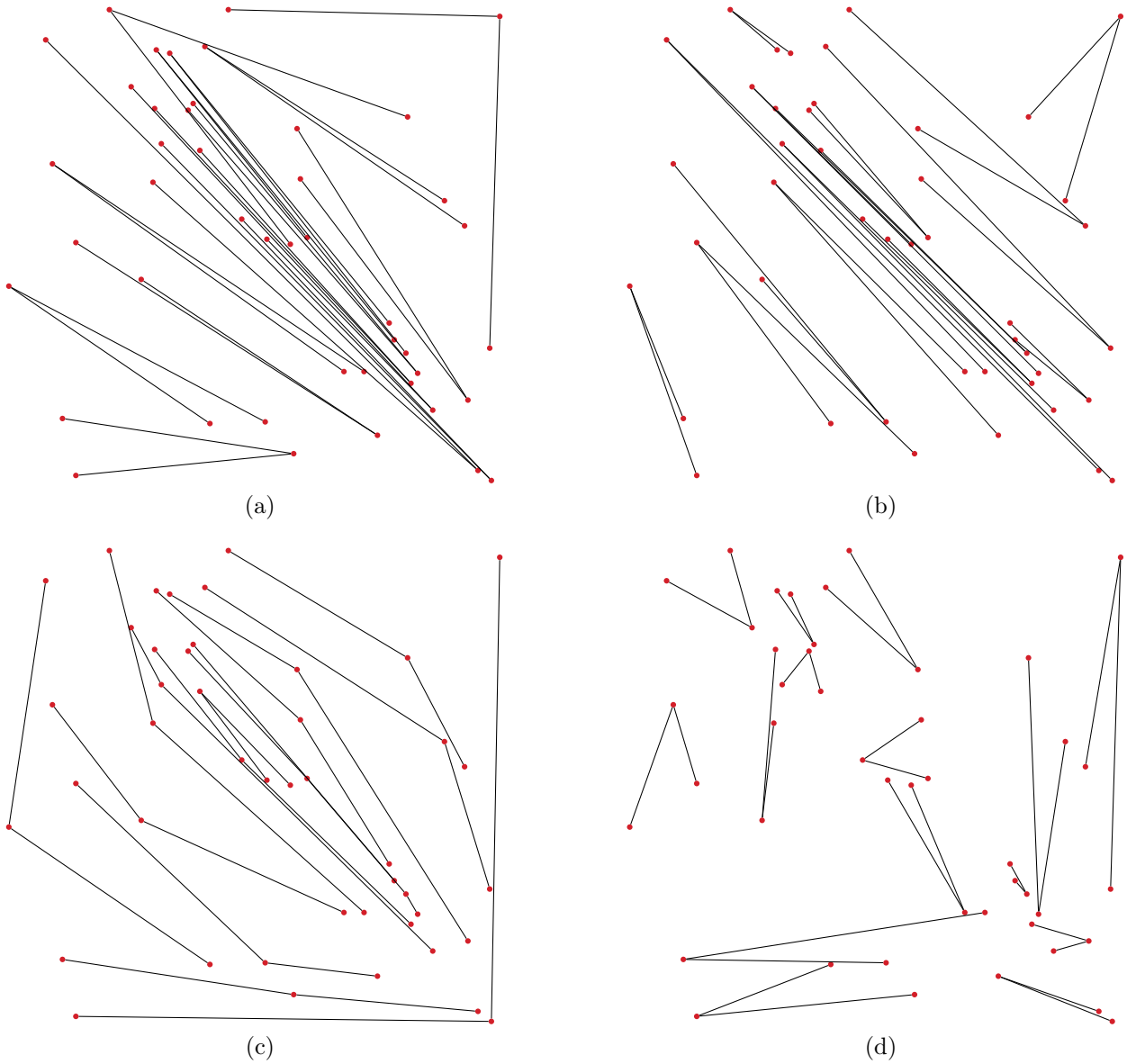
(a) (b) (c) (d)

Figure C.10: Instance 42b



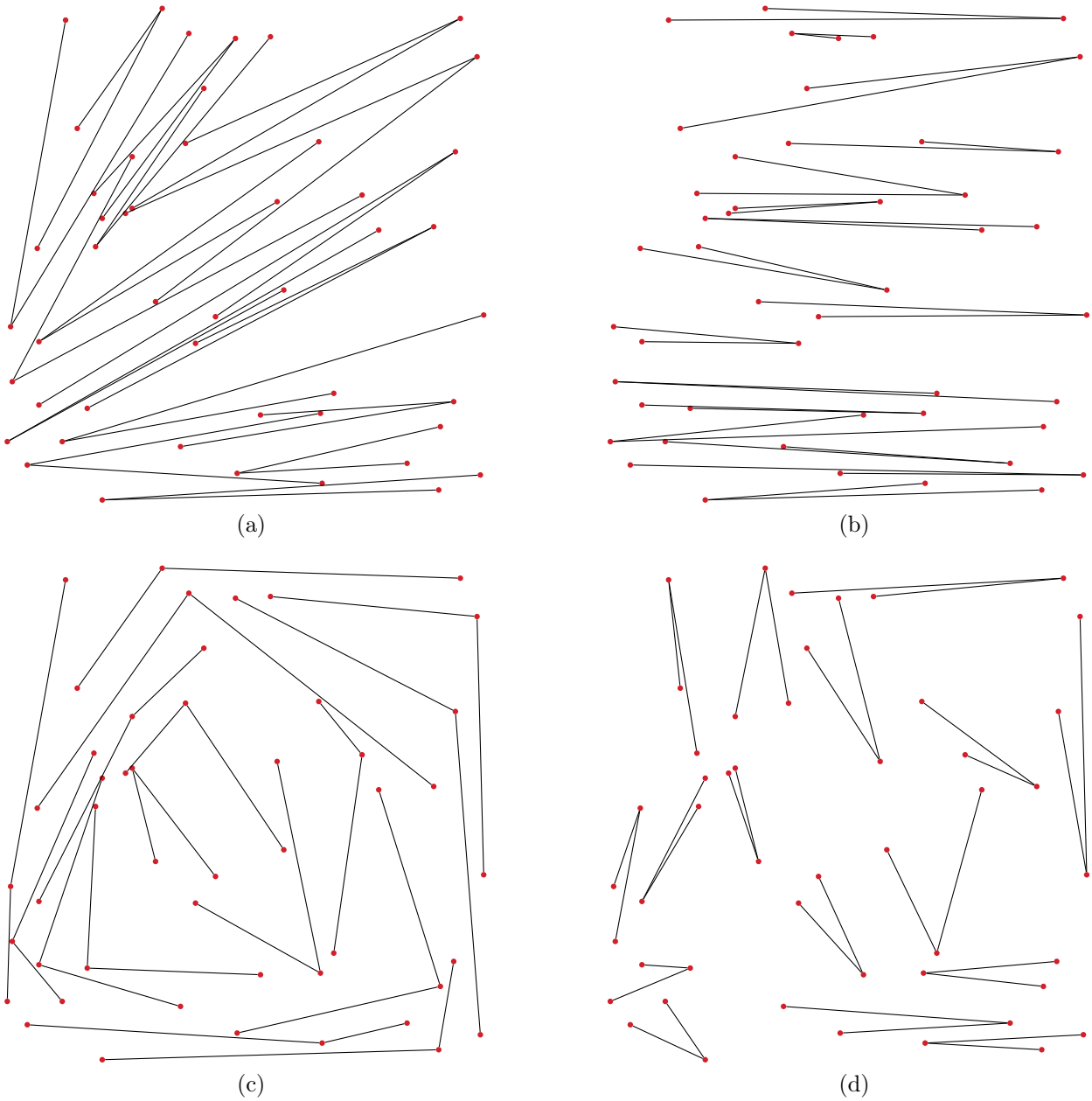
(a) (b) (c) (d)

Figure C.11: Instance 45a



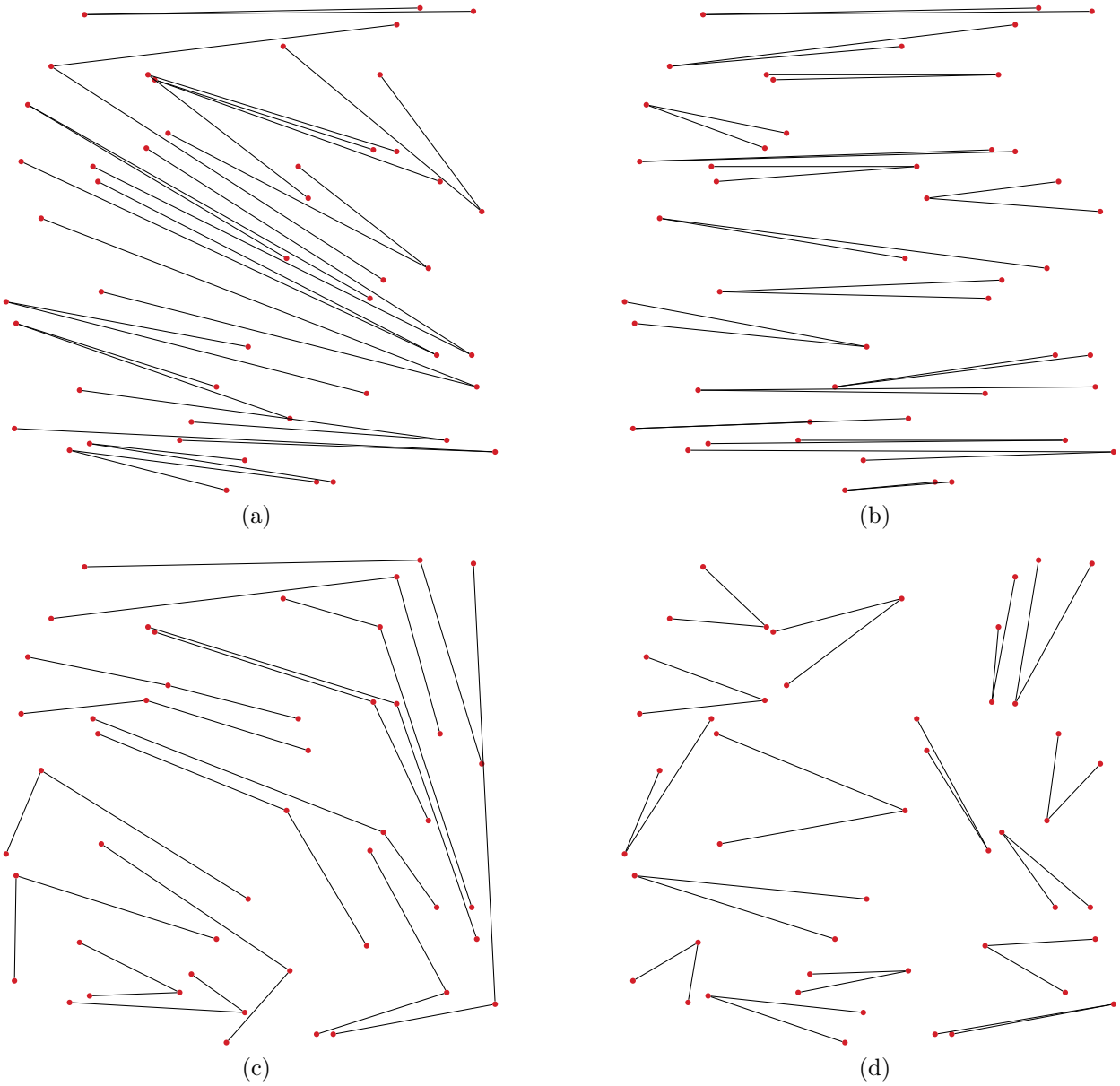
(a) (b) (c) (d)

Figure C.12: Instance 45b



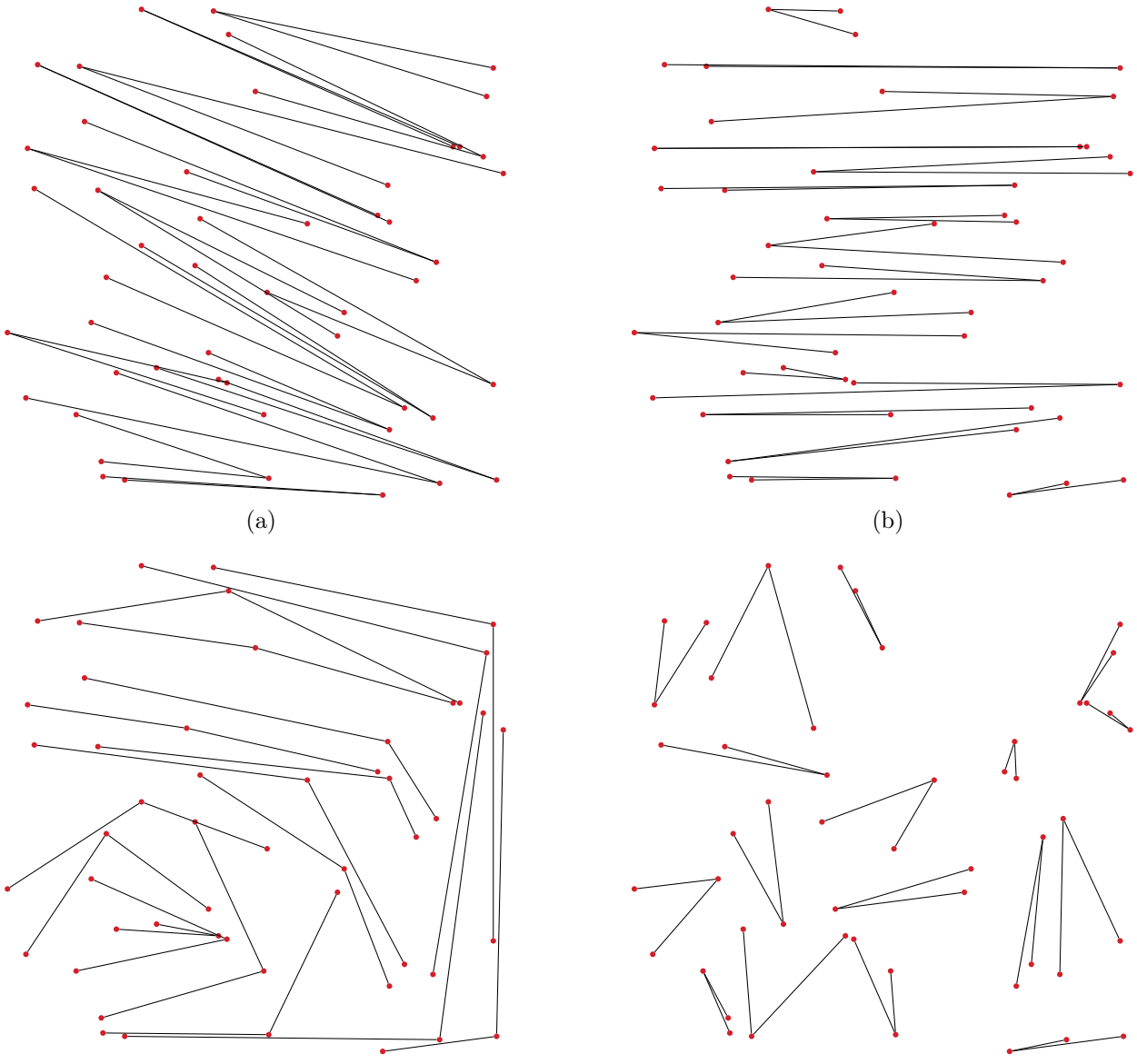
(a) (b) (c) (d)

Figure C.13: Instance 48a



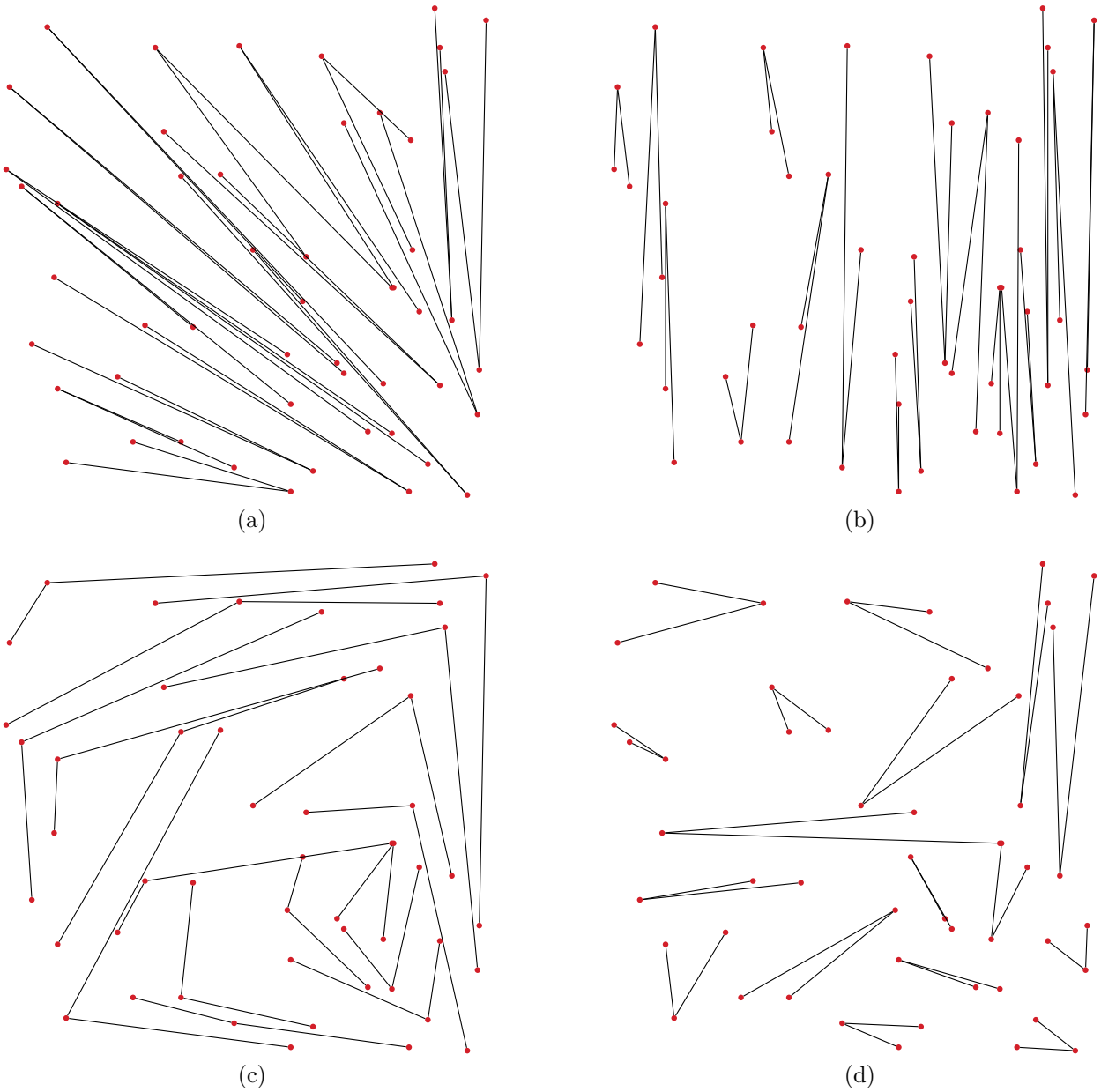
(a) (b) (c) (d)

Figure C.14: Instance 48b



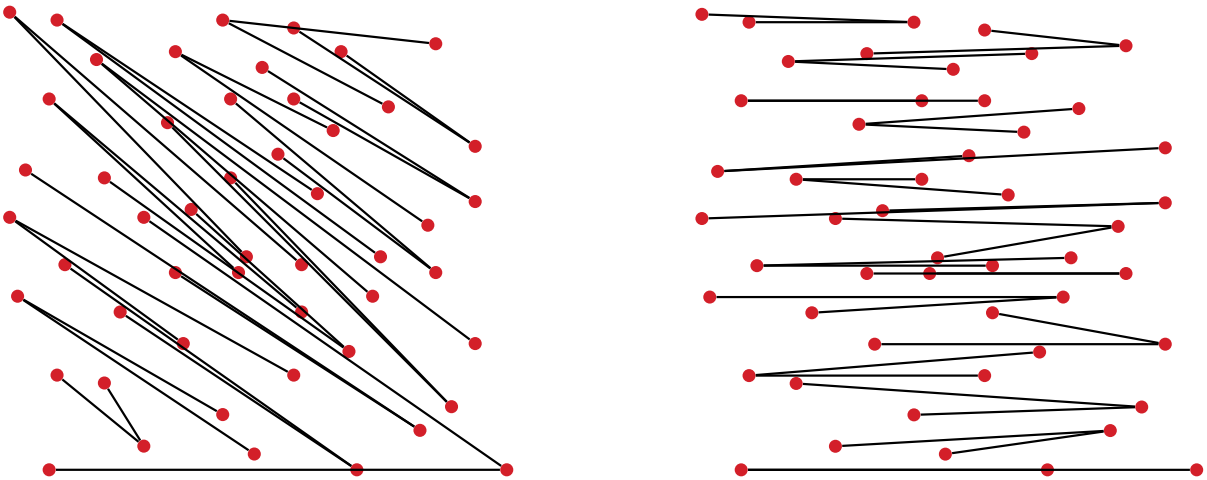
(a) (b) (c) (d)

Figure C.15: Instance 51a

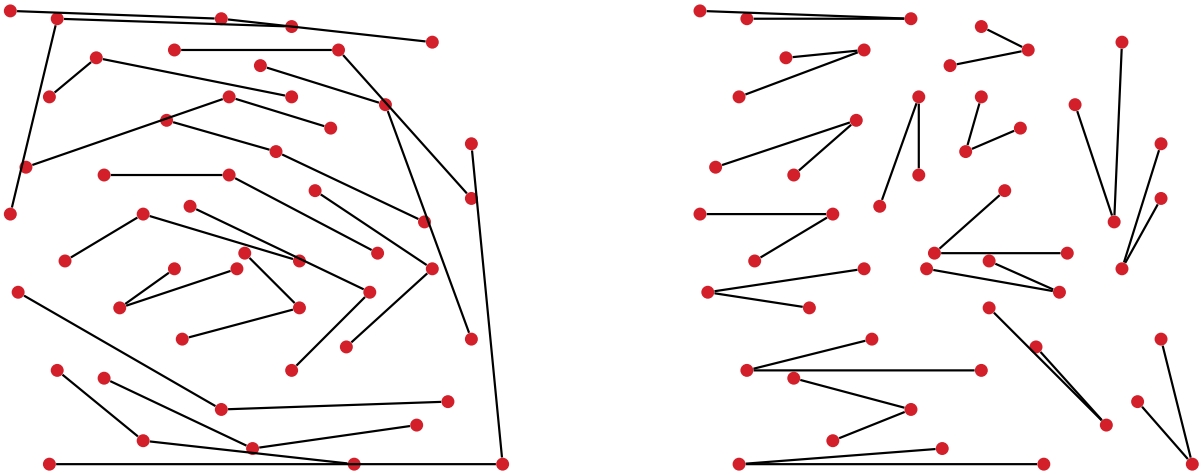


(a) (b) (c) (d)

Figure C.16: Instance 51b



(a) (b)



(c) (d)

Figure C.17: Instance eil51

Bibliography

[1] N. Alon, S. Rajagopalan, and S. Suri. Long non-crossing configurations in the plane. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, SCG '93, pages 257–263, New York, NY, USA, 1993. ACM.

[2] Y. Crama, A. G. Oerlemans, and F. Spieksma. *Production planning in automated manufacturing*. Springer-Verlag, 1994.

[3] Y. Crama and F. C. Spieksma. Approximation algorithms for three-dimensional assignment problems with triangle inequalities. *European Journal of Operational Research*, 60(3):273 – 279, 1992.

[4] A. Dumitrescu and C. D. Tóth. Long non-crossing configurations in the plane. *Discrete & Computational Geometry*, 44(4):727–752, 2010.

[5] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[6] Gurobi Optimization. Gurobi optimizer reference manual. `http://www.gurobi.com/`, June 2015.

[7] R. Hassin and S. Rubinstein. An approximation algorithm for maximum packing of 3-edge paths. *Information Processing Letters*, 63(2):63–67, 1997.

[8] M. Johnsson, G. Magyar, and O. Nevalainen. On the Euclidean 3-matching problem. *Nordic Journal of Computing*, 5(2):143–171, 1998.

[9] D. G. Kirkpatrick and P. Hell. On the complexity of general graph factor problems. *SIAM Journal on Computing*, 12(3):601–609, 1983.

[10] G. Magyar, M. Johnsson, and O. Nevalainen. On the exact solution of the Euclidean three-matching problem. *Acta Cybernetica*, 14(2):357–376, 1999.

[11] G. Magyar, M. Johnsson, and O. Nevalainen. An adaptive hybrid genetic algorithm for the three-matching problem. *Evolutionary Computation, IEEE Transactions on*, 4(2):135–146, Jul 2000.

[12] O. N. Mika Johnsson, Timo Leipälä. Determining the manual setting order of components on PC boards. *Journal of Manufacturing Systems*, 15(3):155 – 163, 1996.

[13] F. C. Spieksma and G. J. Woeginger. Geometric three-dimensional assignment problems. *European Journal of Operational Research*, 91(3):611 – 618, 1996.

[14] R. Tanahashi and C. Zhi-Zhong. A deterministic approximation algorithm for maximum 2-path packing. *IEICE Transactions on Information and Systems*, 93(2):241–249, 2010.

[15] P. J. M. Van Laarhoven and W. H. M. Zijm. Production preparation and numerical control in PCB assembly. *International Journal of Flexible Manufacturing Systems*, 5(3):187–207, 1993.

[16] G. Vazquez Casas, R. A. Castro Campos, M. A. Heredia Velasco, and F. J. Zaragoza Martínez. A triplet integer programming model for the Euclidean 3-matching problem. In *12th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE 2015)*, pages 1–4, Oct 2015.

[17] G. Vazquez Casas, R. A. Castro Campos, M. A. Heredia Velasco, and F. J. Zaragoza Martínez. Integer programming models and heuristics for non-crossing euclidean 3-matchings. In *Numerical and Evolutionary Optimization NEO 2016*, volume 731 of *Studies in Computational Intelligence*, pages 125–140, 2017.